VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Jazyk VPL a jeho využití v programování robotů

Usage of VPL language in robots' programming

2010 Jiří Melcr

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2010 .....

Abstrakt

Práce je zaměřena na seznámení s možnostmi Microsoft Robotics Development Studia, zejména pak na jazyk VPL a jeho využití pro programování robotů. Jednotlivé pojmy a postupy jsou nejprve teoreticky vysvětleny a následně je formou příkladů a úloh také předvedeno jejich praktické využití. Zvolenou cílovou platformou pro demonstraci vývoje robotických aplikací je model sestavený ze stavebnice LEGO® MINDSTORMS® NXT a jeho simulace ve virtuálním prostředí Microsoft Robotics Development Studia. Čtenář by měl být po přečtení tohoto textu schopen orientace v problematice i samostatné tvorby vlastních programů v jazyce VPL.

Klíčová slova: VPL, Visual Programming Language, Microsoft Robotics Development Studio, MRS, MRDS, NXT, Lego, Mindstorms, CCR, DSS, LaTeX, bakalářská práce

Abstract

This work aims to introduce the possibilities of Microsoft Robotics Development Studio, especially the usage of VPL language in robots' programming. First the individual terms and methods are theoretically explained and subsequently its practical application is demonstrated by means of examples and tasks. To demonstrate the development of robotics applications a model assembled from the LEGO® MINDSTORMS® NXT kit and its simulation in the virtual environment of Microsoft Robotics Development Studio have been used. After reading this text the reader should be familiar with the problem and should be able to create his/her own programmes in the VPL language.

Keywords: VPL, Visual Programming Language, Microsoft Robotics Development Studio, MRS, MRDS, NXT, Lego, Mindstorms, CCR, DSS, LaTeX, bachelor thesis

Seznam použitých zkratek a symbolů

BT	–	Bluetooth
CCR	–	Concurrency and Coordination Runtime
CD	–	Compact Disc
CLR	–	Common Language Runtime
CPU	–	Central Processor Unit
DSS	–	Decentralized Software Services
DSSP	–	Decentralized Software Services Protocol
FW	–	Firmware
GDD	–	Generic Differential Drive
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
HW	–	Hardware
I/O	–	Input / Output
LED	–	Light Emitting Diode
MB	–	Megabyte
MRDS	–	Microsoft Robotics Development Studio
MRS	–	Microsoft Robotics Studio
MSDN	–	Microsoft Developer Network
NXT-G	–	NXT-Graphical – grafický programovací jazyk
OS	–	Operační systém
PC	–	Personal Computer
PDA	–	Personal Digital Assistant
SOAP	–	Simple Object Access Protocol
SPL	–	Simulation Programming Language
SQL	–	Structured Query Language
SW	–	Software
USB	–	Universal Serial Bus
VPL	–	Visual Programming Language
V/V	–	Vstupně / výstupní
VS	–	Visual Studio
WPF	–	Windows Presentation Foundation
XML	–	Extensible Markup Language
XSLT	–	Extensible Stylesheet Language Transformations

Obsah

Seznam obrázků

1 Úvod

Práce je zaměřena na seznámení s možnostmi Microsoft Robotics Development Studia (dále jen MRDS), zejména pak na jazyk VPL a jeho využití pro programování robotů.

Pod pojmem robot si můžeme představit cokoli od mlýnku na maso, přes svařovací automat v továrně na automobily, až po inteligentní bytost nahrazující člověka ve sci-fi románech. Dnes jsou realitě blíž spíše první dvě zmíněné představy, které lze charakterizovat jako stroje s více či méně přesně stanovenými úkoly, které vykonávají v reálném čase podle známého algoritmu a většinou na jednom místě nebo ve velmi omezeném prostoru. Jejich hlavním úkolem je nahradit člověka tam, kde ho předčí silou, rychlostí, přesností, vytrvalostí nebo odolností vůči vnějším vlivům. Charakteristická je dnes také jejich specializace na určitou úzce vymezenou činnost. To umožňuje, aby se v různých segmentech trhu prosazovaly různé firmy se svými vlastními proprietálními řešeními, založenými většinou na dlouholetém evolučním vývoji.

V jistém ohledu to připomíná situaci na trhu s výpočetní technikou v 70. letech minulého století, který byl zpočátku rovněž poměrně malý, segmentovaný a úzce specializovaný. Bylo obvyklé, že výrobce hardwaru vyvíjel a implementoval také vlastní software a komunikace výrobků různých firem byla v lepším případě obtížná, v horším dokonce nemožná. Totéž platilo pro přenositelnost SW mezi zařízeními různých výrobců. Postupně docházelo ke standardizaci rozhraní a komunikačních protokolů a výsledkem je dnešní stav, kdy je, až na skutečně úzce specializované segmenty trhu, standardizace a platformová nezávislost téměř nutností. Tyto standardy může definovat nějaká mezinárodní organizace de jure, ale často k ní dochází živelně, de facto, tím, že si nějaký produkt získá takové místo na trhu, že ho ostatní výrobci nemohou porazit ani ignorovat a musejí se mu přizpůsobit. Je to tedy jakási obdoba přirozeného výběru v přírodě, kdy vítězí silnější, většinou bez ohledu na morální pojmy jako lepší, horší.

Zřejmě podobná úvaha může stát za rozhodnutím firmy Microsoft vstoupit na trh s aplikacemi pro řízení robotů. Jak bylo zmíněno výše, dnes jsou ještě roboty relativně statickými stroji s minimální interakcí s okolím. S rostoucími výkony hardwaru na jednotku hmotnosti a plochy se však blíží doba, kdy budou roboty (a možná bude případnější přejít do životného rodu roboti) skutečně mobilní, schopní komunikace s lidmi i mezi sebou navzájem a budou tedy vyžadovat nějakou platformu, na které si budou „rozumět“. S jejich masovějším rozšířením také přijde hromadná průmyslová výroba a zřejmě i oddělení produkce HW a SW. A právě toto může být příležitost pro firmu, která má již dnes dostatečné kapacity finanční i lidské na investice do zatím pravděpodobně ztrátového podniku.

Když se rozhlédneme po zahraničních, ale dnes i domácích, univerzitách a technických vysokých školách, na každé už nalezneme tým robotiky. Motorem výzkumu jsou zde například turnaje robotů ve fotbale, závody v přesnosti, orientaci či vytrvalosti. Je zde tedy velký potenciál mladých studentů, vědeckých pracovníků a vývojářů, které lze novým produktem oslovit a vychovat si tak budoucí zákazníky.

Podle jednoho z iniciátorů vstupu Microsoftu na pole robotických technologií Tandy Trowera[1] se rozhodujícím impulzem stala v roce 2004 série setkání s představiteli robotické komunity, kteří společně, bez ohledu na oblast, které se jejich projekty týkaly, vyzvali Microsoft, aby svými zkušenostmi přispěl k rozvoji tohoto odvětví techniky.

Autor tohoto textu se neodvažuje předjímat budoucí vývoj, přesto věří, že jde o krok správným směrem a že informace nabyté studiem pramenů, a snad srozumitelně včleněné do této práce, bude moci v budoucnu uplatnit i v praxi. Současně dostupnost téměř profesionálního hardwaru v podobě stavebnice LEGO® MINDSTORMS® NXT, na kterém lze bez větších „strojařských" znalostí testovat vyvíjené aplikace, umožňuje podílet se na rozmachu nového odvětví téměř každému zájemci.

Bohužel u nás dosud neexistuje publikace v českém jazyce, která by se svým zaměřením, rozsahem a zejména kvalitou zpracování blížila [1], již lze oprávněně považovat za „bibli" vývojářů v prostředí MRDS. Většina škol (zejména středních), využívá k výuce robotiky LEGO® MINDSTORMS® NXT, ovšem nejčastěji společně s nativním jazykem NXT-G a tomu také odpovídá struktura dostupných materiálů. Přesto existují publikace, které se MRDS ve větší či menší míře věnují (např. [2]), na VŠB-TU Ostava pak v předchozích dvou letech [3, 4, 5].

Tato práce si klade za cíl doplnit do mozaiky další dílek a přiblížit jazyk VPL, coby jednu z podstatných součástí MRDS. V jednotlivých kapitolách čtenáře postupně seznámí s produktem Microsoft Robotics Development Studio, na jednoduchých příkladech představí základní filozofii produktu a jeho základy založené na běhovém prostředí CCR a DSS, formou typických úloh demonstruje tvorbu obecných aplikací v jazyku VPL a po stručném popisu stavebnice LEGO® MINDSTORMS® NXT přinese sadu úloh zaměřených na jazyk VPL jako nástroj pro vývoj robotických aplikací. Nedílnou součástí výkladu jsou také obrázky názorně ilustrující probírané téma.

Výklad je určen pro čtenáře bez hlubších znalostí v oboru robotiky, konstrukce strojů, řízení hardwaru apod. a neklade si tedy v tomto směru žádné vstupní podmínky. Detailní informace o stavebnici LEGO® MINDSTORMS® NXT jsou předmětem samostatné kapitoly, přesto je na tomto místě třeba zmínit, že je dle výrobce určena pro děti od 10 let, nevyžaduje žádné nářadí nebo nástroje a čas potřebný k sestavení běžných modelů robotů se pohybuje v řádu desítek minut.

Základní orientace v oblasti programování, programovacích jazyků (zejména C#), algoritmů apod. se u čtenáře tohoto textu předpokládá. Nejde o podmínku nutnou, příklady budou detailně vysvětleny, komentovány a předvedeny, ovšem pro správné porozumění a zejména pro aplikaci přečteného na vlastní projekty je zvládnutí např. principů algoritmizace velkou výhodou. Některé kapitoly, např. 3 či 4, už vyžadují hlubší znalost C#, ovšem jejich studium není pro zbytek textu nezbytné, jsou zde uvedeny především pro vysvětlení celkové koncepce MRDS a jazyka VPL.

Jednotlivé pojmy a postupy jsou nejprve teoreticky vysvětleny a následně demonstrovány na příkladech, jejichž obtížnost s postupným rozšiřováním nabytých vědomostí vzrůstá. Tento text je zaměřen pouze na softwarovou stránku robotických aplikací, proto nejsou součástí výkladu detailní návody na sestavení robota a většina úloh je také pre-

[1] Úvod jeho knihy [1]

zentována ve virtuálním prostředí MRDS, ovšem pro zájemce o praktickou realizaci jsou k dispozici také odkazy na stavební návody použitých modelů.

Jako platforma pro implementaci konkrétních úloh byla v souladu se zadáním bakalářské práce zvolena stavebnice robotů LEGO® MINDSTORMS® NXT firmy Lego®. V případech, kdy je to pro pochopení úloh vhodné, jsou vedle vlastního zdrojového kódu programu zařazeny také ilustrační obrázky robota během provádění úloh ve virtuálním prostředí.

Ačkoli název Visual Programming Language, tedy vizuální programovací jazyk, svádí k domněnce, že jeho zvládnutí je snazší než zvládnutí klasických programovacích jazyků, protože stačí „pospojovat kostičky“, je třeba upozornit, že jde pouze o formu prezentace algoritmu, tedy vlastně syntaxi a nikoli význam (sémantiku) jazyka. Editor jazyka nás sice oprostí od nutnosti psát kvanta textu, ale myšlenková konstrukce musí být na pozadí přítomna vždy a tvůrčí přístup k novým problémům vyžaduje zvládnutí tohoto jazyka stejně jako by to vyžadovalo programování v jakémkoli jiném jazyce. Někdy dokonce narazíme na limity vizuálního programování a pak je dobré vědět, jaké máme další alternativy a jak je můžeme využít. Úlohy, které budeme v této práci řešit, byly navrženy tak, aby ukázaly výhody jazyka VPL, ale také upozornily na možné chyby a jeho omezení.

Z důvodu snazší demonstrace jsou úlohy určeny pro model robota Tribot, jehož věrná vizualizace ve virtuálním prostředí je zdarma dostupná a je tedy možno funkčnost aplikací nejprve ověřit a teprve po odladění sestavit robota a aplikaci provádět v reálném čase. Tento postup klade jistá omezení co do počtu a typů použitých senzorů, ale k vysvětlení principů je plně postačující a případné odlišnosti jsou v textu vysvětleny.

2 Microsoft Robotics Development Studio

2.1 Vývoj MRDS

Už v úvodní kapitole jsme zmínili Tandy Trowera, nyní generálního manažera Microsoft Robotics Group, který stál u zrodu koncepce MRDS a který uvádí rok 2004 jako počátek jeho vývoje. V knize [1], jíž je spoluautorem, se také dozvíme, že první prototyp Microsoft představil v červnu roku 2006 a v prosinci téhož roku uvolnil první „ostrou“ verzi pod názvem Microsoft Robotics Studio 1.0.

V květnu roku 2007 byla vydána nová verze Microsoft Robotics Studio 1.5 a v prosinci téhož roku verze 1.5 'Refresh'.

V listopadu roku 2008 se Microsoft rozhodl přejmenovat produkt na Microsoft Robotics Development Studio 2008.

V době psaní této práce (4. května 2010) je aktuální verzí MRDS 2008 R2, vydaná v červnu roku 2009, a v ní byly také odladěny veškeré zde uvedené příklady.

2.2 Licenční politika

Vývojová prostředí MRS a MRDS byla od samého počátku zaměřena na celou oblast vývoje robotických aplikací, která zahrnuje jak výzkum a výuku na akademické půdě, tak práce amatérských nadšenců, ale i výrobce profesionálních zařízení. Z tohoto důvodu Microsoft už první verzi MRS 1.0 vydal ve dvou variantách, pro komerční i nekomerční použití.

Od verze MRDS 2008 zavedl přehlednější značení produktů a je tedy možné používat MRDS Standard Edition pro komerční aplikace, MRDS Academic Edition pro akademickou obec a MRDS Express Edition pro amatérské nekomerční aplikace.

2.3 Seznámení s produktem

MRDS je již od první uveřejněné verze koncipováno jako komplexní prostředí určené k vývoji, ladění a řízení aplikací pro obecná robotická zařízení. Jak bude dále detailně vysvětleno, tato koncepce umožňuje vytvářet aplikace pro širokou škálu zařízení, která s pojmem robot v běžném smyslu toho slova nemusí vůbec souviset.

Vývojáři postavili MRDS na základech nových programových modelů, jejichž vývoj probíhal souběžně na jiných pracovištích Microsoftu a jejichž původním záměrem bylo umožnit jednoduchý a rychlý vývoj vícevláknových, decentralizovaných a asynchronních aplikací. O těchto modelech pojednávají samostatné kapitoly 3 resp. 4, zde pouze uveďme, že přesně tyto požadavky jsou kladeny i na robotické aplikace schopné řídit více robotů současně a zpracovávat z nich přijaté informace v reálném čase.

Architektura aplikací vyvíjených v MRDS je založena na nezávislých stavebních kamenech nazvaných *services* (dále budeme používat český ekvivalent služby), které mezi sebou komunikují výhradně zprávami a nijak přímo se vzájemně neovlivňují.[2] Tento

[2]Na rozdíl od dnes rozšířeného modelu vícevláknových aplikací nevyužívají služby společnou paměť hostujícího procesu, nevyžadují od programátora implementaci zámků a semaforů pro synchronizaci.

model umožňuje velkou odolnost proti chybám (selhání jedné služby nenaruší běh jiné), jejich nezávislý vývoj a také široké možnosti prezentace jimi předávaných zpráv, neboť ty jsou předávány formou dokumentů a jejich prezentace uživateli je tak oddělena od vlastního kódu služby.

Aplikace tak lze skládat spojováním a kombinací různých služeb a tím dosáhnout požadované funkce bez nutnosti přepisovat celý program. Pokud například požadujeme přidat u některé služby specifické funkce tak, aby byla schopna spolupracovat s nově přidanými funkcemi našeho hardwaru, stačí přepracovat tuto jedinou službu, případně ji vyměnit za jinou, a zbytek programového kódu zůstane nezměněn.

Výše zmíněné výhody zároveň umožňují do značné míry oddělit návrh řídícího SW robota od detailní komunikace s jeho HW. U správně navržené aplikace k jejímu přenosu na HW jiného výrobce postačí, když se vymění služby svázané s daným typem HW za nové.[3] Tím lze dosáhnout velké míry standardizace a platformové nezávislost, protože výrobce HW se postará pouze o dodání příslušných služeb spjatých s jeho výrobkem a návrh nadstavbové aplikace nechá na specializovaném výrobci.[4] Příkladem takového přístupu k tvorbě aplikací jsou tzv. generické služby, o nichž pojednává kapitola 4.7 a detailně pak kapitola 7.2.

Protože MRDS běží nad prostředím .NET, mohli tvůrci zachovat možnost vývoje aplikací v široké řadě programovacích jazyků (C#, C++, Visual Basic, Python) a navíc přidali jazyk nový, nazvaný VPL (Visual Programming Language). Zpřístupnili tak vývoj robotických aplikací i těm, kteří mají s programováním minimální předchozí zkušenosti. Pomocí vizuálního jazyka mohou snadno do tohoto světa proniknout.

Pro snadné ladění aplikací obsahuje MRDS také velmi kvalitní simulátor. Umožňuje definovat jak parametry (fyzikální vlastnosti a chování) existujícího robota, tak i takového, který má být teprve zkonstruován. Podobně lze definovat fyzikální vlastnosti prostředí, ve kterém se robot bude pohybovat, i překážek a objektů v něm.

Ačkoli se autor tohoto textu setkal při sbírání podkladů zejména na internetu s kritikou Microsoftu za nedostatečnou nebo nekvalitní dokumentaci, je třeba konstatovat, že to v žádném případě neplatí o dokumentaci, jež je součástí instalace MRDS. Ta obsahuje obrovskou paletu příkladů ve formě hotových projektů včetně komentářů a odkazů. Podobně velmi propracovaná je i online dokumentace na MSDN [6] a co ocení vývojář v praxi nejvíce, je fórum [7] s aktivní účastí tvůrců MRDS. Výhrady tak lze mít spíše k nedostatku sekundárních pramenů, zejména v tištěné podobě. Zvláště výrazné je to u projektů zaměřených na LEGO® MINDSTORMS® NXT, neboť většina autorů používá k programování robotů nativní prostředí NXT-G a o MRDS se pouze letmo zmíní. Svou roli samozřejmě také hraje relativní „čerstvost“ MRDS a v budoucnu se určitě dočkáme.

[3] Samozřejmě za předpokladu, že nový HW je určen pro plnění podobných úloh, v tomto směru tedy musí být kompatibilní.

[4] Zkušenosti ze světa mobilních telefonů a PDA, kde podobný proces již nějakou dobu probíhá, ukazují, že tato cesta není tak přímá, jak by se mohlo zdát.

2.4 Části MRDS

Jak již bylo zmíněno výše, MRDS je komplexním prostředím, které se skládá z řady společně použitelných, nicméně do jisté míry nezávislých nástrojů. Každý z nich se zaměřuje na jinou fázi vývoje aplikací a zejména u větších projektů se předpokládá také jejich využití různými členy vývojového týmu.

Tento text je primárně zaměřen na využití jednoho nástroje, a to jazyka VPL, proto mu bude věnována největší pozornost v samostatné kapitole 5 a dále pak samozřejmě při řešení praktických úloh. Větší pozornost si zaslouží také vizualizace, která je obecně popsána v následující kapitole, její praktické využití pak v kapitole 7.6. Ostatní části jsou popsány pouze do hloubky nutné pro pochopení souvislostí, případně pro úspěšné zvládnutí dále prezentovaných příkladů.

2.5 Vizualizace

Nástroj vizualizace, začleněný do balíku aplikací MRDS, slouží k demonstraci vytvořených aplikací na robotech ve virtuálním prostředí, které věrně simuluje reálné prostředí, ve kterém má být robot nasazen. K tomuto účelu má programátor (designér) k dispozici sadu nástrojů[5] pro modelování robota samotného (jeho jednotlivých částí a jejich fyzikálních vlastností) i prostředí okolo něj. Je možné návrhy provádět „na zelené louce“ nebo využít či rozšířit již vytvořené modely, kterých je už v základní distribuci MRDS dostatek.

Ačkoli je vizualizace velmi silným nástrojem MRDS a pro některé může být i hlavním motivem jeho využití, my jí v této práci věnovat podrobněji nebudeme, protože s jazykem VPL jako takovým bezprostředně nesouvisí. Simulace ovšem budeme hojně využívat k prezentaci úloh vytvářených ve VPL, proto se v kapitole 7.6 s prostředím vizualizace alespoň částečně seznámíme.

2.6 Dokumentace

Samotná instalace MRDS obsahuje také velmi kvalitní dokumentaci ve formě referenční příručky všech části MRDS i tutoriálů pro zvládnutí základů práce v prostředí MRDS, včetně ukázkových příkladů se zdrojovými kódy. V dalším textu se budeme na tuto dokumentaci často odvolávat a je na čtenáři, aby si v ní informace, které zde nemůžeme podrobně popisovat, našel sám. Soubory dokumentace se nacházejí v podadresáři *documentation* instalace MRDS, ukázkové aplikace pak v podadresáři *samples*.

2.7 Systémové nároky

Jediným systémovým nárokem, který firma Microsoft uvádí na svých stránkách, je typ operačního systému. MRDS je tak podporováno pouze pro Windows XP a Windows Vista. Autor tohoto textu nicméně velkou většinu příkladů a úloh odladil na stroji se systémem Windows 7 a nezaznamenal žádné problémy.

[5] Nedávno se objevil nový skriptovací jazyk SPL jako nadstavba pro vytváření simulací v MRDS. Podrobné informace s množstvím praktických příkladů lze nalézt na [8].

Pro tvorbu, editaci a kompilaci služeb přímo v jazyce C# je nutno zvlášť doinstalovat libovolnou verzi Windows Visual Studia, autor použil VS 2008 Standard Edition.

Z hlediska hardwarových nároků by měl pro správnou funkci stačit jakýkoli stroj splňující minimální nároky pro běh daného OS. Současně je na tomto místě třeba upozornit, že nejvyšší hardwarové nároky MRDS klade na grafickou kartu a paměť počítače zejména z důvodu hladkého běhu vizualizací, z tohoto důvodu by minimální doporučená konfigurace nemusela být dostatečná. Pokud ovšem čtenář nehodlá tuto část MRDS využívat, případně se spokojí s nižší kvalitou výstupu, lze aplikace plnohodnotně vyvíjet i na podstatně slabších strojích.

Autor pracoval na notebooku HP Compaq 6910p s Intel® Core™ 2 Duo 2,2 GHz, 1GB (později 3GB) RAM a 80 GB HDD s OS Windows XP (později Windows 7) v české lokalizaci se všemi dostupnými aktualizacemi a záplatami.

2.8 Instalace MRDS

Jak je v poslední době obvyklé, nabízí Microsoft možnost instalace online, kdy si v jejím průběhu stahuje potřebné komponenty z vlastního serveru, nebo offline, kdy jsou všechny již součástí instalačního balíku.

V současnosti (květen 2010) je aktuální verze MRDS 2008 R2, která je k dispozici ve verzích Express Edition, Standard Edition a Academic Edition (ta byla také použita autorem).

Poznámka 2.1 Všechny verze mimo Standard jsou dostupné přímo jako R2, tato jediná pouze jako upgrade verze předchozí. Je tedy třeba nejprve nainstalovat MRDS 2008 Standard Edition a následně upgrade na R2. Instalace upgrade ponechá v systému i předchozí verzi, obě tak lze používat nezávisle (platí minimálně pro aktuální verzi R2). Je třeba ovšem dodržet licenční politiku, tzn. upgradovat lze pouze takovou verzí, která odpovídá z hlediska licence verzi původní.

Během instalace máme možnost zvolit cílový adresář, do kterého se hlavní část MRDS nainstaluje. Jako výchozí nabízí instalátor *C:\ Program Files\Microsoft Robotics Dev Studio 2008 R2*. Je vhodné si umístění dobře rozmyslet, protože tento adresář (a jeho podadresáře) používá MRDS mimo jiné k ukládání všech zkompilovaných služeb, ukázkových aplikací, editovaných prostředí a ve výchozím nastavení i VPL projektů. Nejenže tak spotřebovává místo na disku, ale navíc by měl být uživateli přístupný pro případné editace.

2.9 Instalace SimplySim

V některých úlohách je jako volitelná možnost pro ověření aplikací doporučeno také virtuální prostředí společnosti SimplySim. O vizualizaci jako takové pojednává samostatná kapitola 7.6 a o odlišnostech a výhodách uvedeného nástroje pak 7.6.3. Zde pouze popíšeme postup instalace, abychom prostředí mohli následně používat. Pokud ovšem čtenář toto prostředí využívat nehodlá, instalaci provádět nemusí.

Instalační balík tvoří pouze jeden spustitelný soubor *NXT-MSRDS-R2.exe*[6] o velikosti cca 5,5 MB. Ten lze získat ze stránek společnosti SimplySim [9] nebo z přiloženého CD.

Jedinou nezbytnou podmínkou pro správnou funkci virtuálního prostředí je umístění instalovaných souborů do podadresáře instalace MRDS, aby se tak staly pro prostředí MRDS viditelnými. Rovněž v případě budoucího upgrade je třeba postupovat ve stejném pořadí, tzn. nejprve instalovat MRDS a teprve poté prostředí SimplySim.

[6] Je k dispozici instalace i pro předchozí verzi MRDS, ale neobsahuje tolik funkcí, proto můžeme doporučit upgrade na aktuální.

3 Concurrency and Coordination Runtime (CCR)

Tato kapitola si klade za cíl představit a vysvětlit základní stavební kámen všech aplikací vyvíjených pod MRDS a je určena zejména pro čtenáře s poměrně dobrou znalostí jazyka C# a implementací I/O operací v něm. Přestože jsou jednotlivé vlastnosti demonstrovány na ukázkách právě v jazyce C#, jsou také velmi podrobně popsány i slovně a alespoň pro rámcovou představu o celé koncepci MRDS je tedy její prostudování doporučeno i čtenářům bez výše zmíněných znalostí.

3.1 Synchronní komunikace

Běžné programovací techniky používají pro komunikaci s externími procesy[7] synchronní přístup. Ten je charakteristický tím, že aplikace vykonává programový kód sekvenčně a po odeslání požadavku na provedení operace externím procesem čeká na odpověď, teprve po jejím obdržení pokračuje v provádění svého kódu (viz obr. 1). Pokud je tedy operace časově náročná, může docházet k dlouhým stavům nečinnosti vlastní aplikace. To se pak projevuje například celkovým zpomalením jinak nenáročných operací, špatnou reakcí uživatelského rozhraní apod.

Výhodou tohoto postupu je velmi dobrá přehlednost zdrojového kódu, neboť je v každém okamžiku zřejmé, jak a kde bude činnost programu pokračovat. Proto je také tento přístup k programování historicky starší.

Nevýhody jsou z výše popsaného zřejmé. Program stráví příliš mnoho času čekáním (jeho délka se navíc může měnit), může vykonávat pouze jednu část programového kódu v jeden okamžik, přestože má dostatek času, který by mohl využít k provádění jiných částí kódu (typicky vykreslování, komunikace s dalšími procesy apod.).

3.2 Asynchronní komunikace

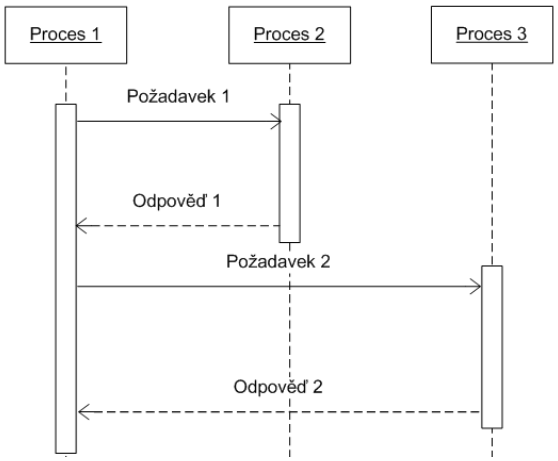
Výše popsané nedostatky lze odstranit použitím asynchronní komunikace mezi procesy. Její princip je znázorněn na obrázku 2. Z něj je patrný zásadní rozdíl proti synchronnímu modelu. Aplikace již nečeká na odpověď volaného procesu, ale dál pokračuje v činnosti. Teprve v okamžiku, kdy odpověď dorazí, provede tu část kódu, která s přijatou odpovědí souvisí.[8]

Výhody jsou opět zřejmé, v podstatě řeší všechny popsané nevýhody synchronního modelu. Asynchronní model komunikace tak umožňuje psát efektivnější kód schopný maximálně využít dostupné zdroje.

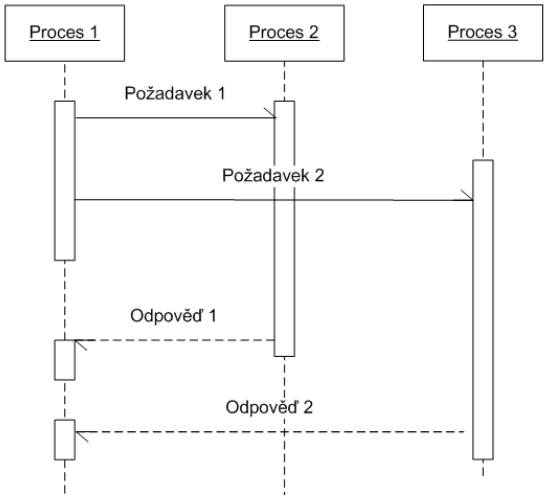
Přesto i tento model má své nevýhody. Takto napsaný kód je daleko méně přehledný, protože obslužné procedury mohou být ve zdrojovém kódu umístěny zcela nezávisle na proceduře volající externí proces. Při analýze kódu je tedy potřeba sledovat nejen hlavní linii programu, ale také všechny obslužné procedury, které by se v danou chvíli mohly

[7] Procesem zde rozumíme např. jinou aplikaci, I/O služby OS apod.

[8] Tato část kódu se označuje jako CallBack procedura, protože je vyvolána odpovědí jiného procesu na náš požadavek – tedy obdoba telefonického zpětného volání jak jej známe z běžného života.



Obrázek 1: Sekvenční diagram synchronní komunikace procesů



Obrázek 2: Sekvenční diagram asynchronní komunikace procesů

provádět. Daleko složitější je rovněž ladění takového programu, protože už nemůžeme krokovat po jednotlivých příkazech.

Dále se vyskytuje problém, který synchronní model neznal, a to koordinace a synchronizace. Pokud totiž odešleme několik požadavků na různé externí procesy za sebou, aniž bychom vždy čekali na odpověď na požadavek předchozí, může se stát (a dokonce je to i výraznou předností tohoto modelu), že odpovědi dorazí v jiném pořadí, než byly požadavky odeslány. Potom musíme zajistit, aby obslužné procedury byly buď zcela nezávislé, nebo aby byly schopny zjistit, zda už dorazila odpověď z procesu, s kterým jejich činnost souvisí. Představme si například, že chceme odeslat zprávu emailem, současně ji uložit do SQL databáze a následně zobrazit uživateli informaci o úspěšném provedení akce. Pokud tento úkol budeme řešit asynchronně, odešleme požadavky na odeslání

mailu i jeho uložení současně. V proceduře zavolané po dokončení každé akce pak musíme nastavit nějaký příznak, abychom tak dali najevo, že tato část úkolu byla provedena. Pokud obslužná procedura jednoho požadavku zjistí, že druhý příznak již byl nastaven, může informovat uživatele.

Podobně musíme řešit i ošetření chybových stavů. Může se stát, že některá operace selže a my musíme rozhodnout o dalším postupu, protože podobně jako v předchozím případě může výsledek jedné operace ovlivnit jinou. Rovněž musíme počítat s možností současných požadavků na čtení nebo zápis do proměnných nebo objektů a musíme tedy řešit i jejich zamykání a priority v přístupech k nim.

Tématy souvisejícími s asynchronním přístupem k programování paralelních aplikací (synchronizace procesů, sdílení zdrojů atd.) se zabývá celá řada studií a algoritmických postupů. Protože jde o velmi dobře vědecky zpracovanou, ale náročnou oblast, nebudeme se jí zde dále věnovat. Toto téma, speciálně s ohledem na jazyk C#, je velmi dobře popsáno třeba v [10].

Je tedy zřejmé, že efektivita běhu programu si bere daň ve formě náročnější tvorby kódu a jeho větší složitosti. Ačkoli většina moderním programovacích jazyků a vývojových prostředí již obsahuje nástroje usnadňující vývoj asynchronně orientovaných programů, stále zůstává značná část odpovědnosti na programátorovi. Nutnost neustále mít na paměti, že v kterémkoli okamžiku běhu programu může dojít k chybě některého asynchronního procesu, a tedy množství podmínek a vzájemných kontrol mezi procesy, často vede k tomu, že výsledné programy nejsou zdaleka tak efektivní jak by být mohly při optimálně navržené koordinaci.

3.3 Koncepce CCR

Právě snaha o zjednodušení návrhu programového kódu při zachování jeho výkonnosti vedla ke zrodu CCR. Autoři se snažili vzít to nejlepší z obou výše zmíněných modelů a vytvořit programátorovi takové podmínky, aby se mohl soustředit na hlavní účel svého kódu a mohl se přitom spolehnout na to, že bude dostatečně efektivní použitím asynchronního modelu a současně přehledný a robustní, neboť odpovědnost za koordinaci a synchronizaci procesů převezme CCR. Sami autoři CCR uvádějí[9], že jejich ambicí nebylo vymyslet nový programovací jazyk nebo nové algoritmy pro paralelizaci procesů, ale využít již dostupné možnosti prostředí .NET a nabídnout knihovnu, která na jedné straně poskytne programátorovi sadu velmi jednoduchých a snadno použitelných metod[10] a na straně druhé zajistí jejich paralelní asynchronní zpracování více procesy, přičemž se sama postará o jejich koordinaci a synchronizaci. Všechny dále zmíněné postupy tedy lze již dnes realizovat i přímo prostřednictvím .NET, ale určitě ne tak přehledně a většinou ani efektivně.

Poznamenejme ještě, že CCR nesouvisí přímo a pouze s programováním robotických aplikací, ačkoli MRDS je prvním produktem, který jej využívá, ale je univerzálně použitel-

[9] Přímo z úst George Chrysanthakopoulose, jednoho z autorů CCR, si tato slova může zájemce poslechnout v jednom z online tutoriálů na internetu [11]

[10] Případně celých doporučených postupů pro řešení typických problémů.

ným modelem pro vývoj libovolných aplikací v prostředí .NET.[11] Jeho výhody vyniknou samozřejmě zvláště tam, kde požadujeme rychlou a efektivní spolupráci velkého počtu procesů (vláken), a to je i případ typických robotických aplikací.[12]

Poznámka 3.1 Ačkoli nová verze .NET 4.0 obsahuje prvky paralelismu, nejedná se o zahrnutí CCR do jádra .NET, naopak tvůrci (viz [13]) CCR počítají s využitím těchto prvků a s dalším vývojem směrem k MRDS 3.0.

V následující části si popíšeme a vysvětlíme jednotlivé dílky mozaiky CCR a uvidíme tak, na čem je založen jazyk VPL, jenž je hlavním předmětem zájmu této práce. Vzhledem k tomu, že pro naše účely nepotřebujeme zacházet do přílišných detailů, bude výklad problematiky poněkud zhuštěný. Zájemci o hlubší studium tohoto zajímavého přístupu k tvorbě asynchronních aplikací nalezenou dostatek zdrojů v seznamu doporučené literatury. Za základní lze považovat zejména tutoriál, jenž je součástí dokumentace k MRDS, jehož osnovu, byť ve zjednodušené podobě, dodržuje i následující část textu[13], a také výborná kniha [1].

Princip programovacího modelu CCR je založen na velmi volně vázaných částech kódu, které spolu komunikují výhradně prostřednictvím zpráv. To usnadňuje jejich koordinaci, ošetřování chybových stavů a rozdělování systémových prostředků mezi ně.

CCR využívá několik základních primitiv, které si nyní popíšeme.

3.3.1 Port

Jde o základní prvek CCR, prostřednictvím kterého mezi sebou jednotlivé komponenty kódu komunikují. Můžeme se na něj dívat jako na dvě paralelní fronty, z nichž jedna uchovává zprávy a druhá jejich příjemce. Jde o generickou třídu, takže můžeme přesně specifikovat typy, které může fronta obsahovat, a ošetřit tak případné chyby v návrhu už při překladu.

```
Port<string> portString = new Port<string>();

//vložení dvou zpráv do portu
portString.Post("Zkušební text 1-1");
portString.Post("Zkušební text 1-2");

Console.WriteLine(portString.ToString()); //port zobrazí v čitelné podobě svůj obsah
Console.WriteLine(portString.ItemCount); //ukáže počet nepřevzatých zpráv
Console.WriteLine(portString.Test()); //vytáhne z portu první nepřečtenou zprávu (a odstraní ji)
Console.WriteLine(portString); // implicitně provede Test()
```

Výpis 1: DemoCCR.cs - demoPort()

[11] Microsoft poskytuje samostatný vývojářský balík CCR i mimo MRDS, je volně dostupný na stránkách firmy [12].

[12] Dalšími typickými oblastmi použití, zejména ve spojení s dále popsaným DSS, jsou například webové služby, databázové aplikace apod.

[13] Zdrojové kódy dále uváděných příkladů, stejně jako aplikace, která umožňuje jejich odzkoušení, jsou součástí CD.

Port umožňuje vkládání zpráv a následně jejich zpracování příjemcem. Na příkladu (výpis 1) vidíme, že zprávy lze z fronty vybírat jednotlivě, ale dále uvidíme, že jejich hlavním účelem je okamžité zpracování příjemcem ihned po jejich odeslání na port.

3.3.2 PortSet

V běžných aplikacích se předpokládá, že portů budeme používat více. Abychom nemuseli vždy specifikovat každý zvlášť, včetně datového typu, který je schopen pojmout, máme k dispozici další primitiv PortSet. Je opět generický a jak vidíme na příkladu (výpis 2), lze jím velmi jednoduše definovat několik portů pro různé typy najednou.

```
PortSet<string,int> portSet1 = new PortSet<string,int>(); //lze vytvořit portset pro 2-19 typů

//vložení dvou zpráv - každá si najde příslušný port dle typu
portSet1.Post("Zkušební text 2");
portSet1.Post(23);

Console.WriteLine(portSet1.ToString());
Console.WriteLine(portSet1.Ports.Count); //ukáže počet vytvořených portů
//následující postup není běžný (dále si ukážeme lepší), ale ilustruje princip PortSetu
Console.WriteLine(portSet1.P1); //již při psaní kódu se můžeme odkazovat na vytvořené porty
    přímo
Console.WriteLine(portSet1.P0); //zde dostaneme instanci Port<string>
```

Výpis 2: DemoCCR.cs – demoPortSet()

Při odeslání zprávy na takto definovaný PortSet se pak zpráva podle svého typu sama zařadí do příslušného portu.To s sebou přináší zejména lepší čitelnost kódu, jak uvidíme dále.

Protože chceme pro zpracování zpráv použít asynchronní model, tzn., záleží nám na tom, aby se zprávy zpracovávaly paralelně, musíme nějak zajistit, aby se pro každou dvojici zpráva-příjemce vytvořilo vždy nové vlákno, v jehož kontextu se operace zpracování zprávy příjemcem provede. Už zde uvidíme první zřetelnou výhodu proti „ručnímu" kódování asynchronních operací. Zatímco dříve bychom museli vždy definovat, které vlákno má úlohu provést (buď vytvořením nového, nebo jeho vyžádáním z ThreadPoolu), teď pouze specifikujeme podmínky, za jakých se mají vlákna vytvářet a CCR sám zajistí, že budou vytvořena přesně v okamžiku, kdy budou potřeba a posléze také odpovídajícím způsobem ukončena.

3.3.3 Dispatcher

K tomu nám slouží třída Dispatcher. V jejím konstruktoru určíme, kolik vláken se smí vytvářet. Standardně je povoleno jedno vlákno pro každé CPU, v případě jednoprocesorového systému pak vlákna dvě (i pokud je vícejádrové).

3.3.4 DispatcherQueue

Tato třída představuje frontu požadavků na přidělení vlákna ze zásobníku dostupných zdrojů. Tímto zásobníkem může být buď klasický thread pool z CLR, nebo lépe instance třídy Dispatcher, spravující vlákna v CCR.

```
VypisVcetneThreadID("Ukázka použití Dispatcher a DispatcherQueue"); //výpis včetně aktuálního
    vlákna

//vytvoření zdroje vláken
Dispatcher dispatcher = new Dispatcher(
    0, // kolik vláken na CPU (0=1:1 nebo 2 pro jediné CPU)
    "Dispečer 1"
    );

//vytvoření fronty na vlákna
DispatcherQueue taskQueue = new DispatcherQueue( "Fronta 1" ,dispatcher);

//vytvoření dvojice zpráva-příjemce, kterou požadujeme vykonat
Task<string> task = new Task<string>("Zkušební text 3",zprava =>
    { //anonymní metoda, opět přispívá k čitelnosti kódu
        VypisVcetneThreadID(zprava); //pro ověření, že bude vykonána jiným vláknem
    });

//zařazení tasku do fronty na volné vlákno
taskQueue.Enqueue(task);

VypisVcetneThreadID("Konec ukázky - stiskněte klávesu"); //výpis ID aktuálního vlákna
Console.ReadKey();
dispatcher.Dispose(); //ukončení všech běžících vláken

Výpis na konzoli:

Vlákno 10 : Ukázka použití Dispatcher a DispatcherQueue
Vlákno 10 : Konec ukázky - stiskněte klávesu
Vlákno 11 : Zkušební text 3
```

Výpis 3: DemoCCR.cs – demoDispatcherQueue()

V tomto příkladu (výpis 3) vidíme, že můžeme do fronty zařadit instanci třídy Task, kterou tvoří dvojice zpráva – metoda. Zde používáme anonymní metodu v úsporném zápisu, který umožňuje C# 3.0.[14] V okamžiku, kdy ve frontě přijde na náš Task řada, bude mu přiděleno vlákno a v jeho kontextu se metoda provede.

Nyní tedy známe základní prvky CCR a můžeme přistoupit k jejich vzájemnému provázání.

Přidělování vláken funguje tak, že po příchodu zprávy do portu se nejprve zjistí, zda existuje nějaký příjemce. Pokud ano, ověří se, zda je příjemce schopen zprávu zpracovat (odpovídá datovému typu zprávy) a v případě kladné odpovědi se vytvoří instance třídy Task, která obsahuje zprávu a jejího příjemce (metodu provádějící zpracování zprávy).

[14] Jde o syntaxi převzatou z jazyka λ-kalkul

Tato instance se zařadí do fronty DispatcherQueue. Fronta pak postupně přiděluje vlákna jednotlivým taskům a následně se vlákno vrátí pro další použití.

3.3.5 Arbiter

O výše popsanou koordinaci se stará několik tříd, souhrnně zvaných arbitři (arbiters). Název i v češtině (arbitr, rozhodčí) poměrně přesně vystihuje jejich úlohu. Mají rozhodnout, kdo bude jakou zprávu a s jakými závislostmi zpracovávat. Ačkoli můžeme vytvářet jejich instance přímo příslušnými konstruktory, dávají tvůrci k dispozici statickou třídu Arbiter s jejími statickými metodami, které instanci příslušné třídy vytvoří a rovnou nastaví příslušné parametry konstruktoru. Jde vlastně o zapouzdření (wrapper) konstrukcí, které bychom jinak pro plánování a koordinaci museli dělat sami a navíc to opět přispívá k lepší čitelnosti kódu (syntactic sugar).

Jednotlivé třídy arbitrů si probereme podrobně, neboť mají přímou vazbu na viditelné a přímo použitelné vlastnosti a chování prvků v jazyce VPL.

- **Receiver**

  Tato třída sváže příjemce s konkrétním portem. Zajišťuje zavolání metody příjemce při příchodu nové zprávy na port. Můžeme specifikovat, zda bude perzistentní, potom je tento svazek pevný až do jeho explicitního zrušení a příjemce je zavolán pro každou obdrženou zprávu, nebo nebude perzistentní a pak je příjemce zavolán pouze jedenkrát, při příchodu první zprávy. Další zprávy sice bude port přijímat, ale příjemce se o nich již nedozví.

```
VypisVcetneThreadID("Ukázka použití arbitru Receiver"); //výpis včetně aktuálního vlákna

Dispatcher dispatcher = new Dispatcher(vlaken,"Dispečer 2");
DispatcherQueue taskQueue = new DispatcherQueue("Fronta 2", dispatcher);

Port<string> port = new Port<string>();

//svážeme port s frontou vláken a perzistentní instancí třídy Receiver
Arbiter.Activate( //svazek bude ihned aktivní=schopen zpracovávat zprávy
    taskQueue,
    // wrapper pro vytvoření instance Receiver
    Arbiter.Receive(true, port, zprava => VypisVcetneThreadID(zprava))
);
port.Post("Zkušební text 5-1");//okamžitě by se měly zprávy zpracovávat
port.Post("Zkušební text 5-2");
port.Post("Zkušební text 5-3");

VypisVcetneThreadID("Konec ukázky - stiskněte klávesu"); //výpis ID aktuálního vlákna
Console.ReadKey();
dispatcher.Dispose(); //ukončení všech běžících vláken

Výstup na konzoli (pro vlaken=0)
Vlákno 09 : Konec ukázky
Vlákno 13 : Zkušební text 5-1
Vlákno 13 : Zkušební text 5-2
```

```
Vlákno 13 : Zkušební text 5−3

Výstup na konzoli (pro vlaken=4):
Vlákno 09 : Ukázka použití arbitru  Receiver
Vlákno 09 : Konec ukázky − stiskněte klávesu
Vlákno 13 : Zkušební text 5−1
Vlákno 15 : Zkušební text 5−3
Vlákno 14 : Zkušební text 5−2
```

Výpis 4: DemoCCR.cs – demoArbiterReceive(int vlaken)

Na příkladu (výpis 4) vidíme různý výstup pro různě nastavený maximální počet vláken na CPU (viz konstruktor Dispatcher). Protože ukázka běžela na jednoprocesorovém stroji (defaultně pouze dvě vlákna), bylo v prvním případě vlákno č. 9 použito pro hlavní program a pro zpracování všech tří zpráv se muselo postupně použít jedno zbylé vlákno. V případě povolení čtyř souběžných vláken na jednom CPU již mohla být každá zpráva zpracována vlastním vláknem. Pouze na konci se vlákna musela seřadit v přístupu ke společné konzoli terminálu. Současně vidíme, že ačkoli fronta zajišťuje, že bude zpracování zpráv zahájeno v pořadí jejich doručení, nelze totéž říct o dokončení zpracování, pokud probíhá několika vlákny souběžně.

Ještě si na výpise 5 ukažme již dříve avizovaný zjednodušený zápis zpracování různých typů zpráv prostřednictvím PortSet. Metoda Activate totiž jako poslední parametr vezme pole instancí arbitrů[15] – libovolných, můžeme tak logiku zpracování vystavět na jednom místě pro všechny porty i jejich příjemce.

```
PortSet<string,int> portSet = new PortSet<string,int>();

//v jednom kroku vytvoříme instance Receiver pro všechny typy zpráv v portSetu
Arbiter.Activate(taskQueue,
   Arbiter.Receive<string>(true, portSet, zprava => VypisVcetneThreadID(zprava)),
   Arbiter.Receive<int>(true, portSet, zprava => VypisVcetneThreadID(zprava.ToString()))
);
portSet.Post("Zkušební text 6−1");//okamžitě by se měly zprávy zpracovávat
portSet.Post(35);
```

Výpis 5: DemoCCR.cs – demoArbiterReceive2()

- **Choice**

  Třída slouží pro zajištění vykonání pouze jedné větve (mutual exclusive) zpracování zpráv. Na příkladu je demonstrováno toto chování pro PortSet s typy Int32 a String, ale dále uvidíme, že toto řešení je ideální zejména pro ošetření chybových stavů, kdy jedna jediná výjimka (odeslaná jako zpráva na port) zajistí přerušení zpracování ostatních zpráv.

Poznámka 3.2 Příjemci ve větvích, které nemají být vykonány, jsou okamžitě z dalšího příjmu zpráv vyloučeni. Tento proces je atomický, tzn., že nemůže být přerušen

[15] Přesněji všech tříd implementujících rozhraní ITask<>, ale o tom později.

jiným vláknem, a je tedy zajištěno, že bude za všech okolností vykonána pouze jedna jediná větev.

```
Dispatcher dispatcher = new Dispatcher(4, "Dispečer 2");
DispatcherQueue taskQueue = new DispatcherQueue("Fronta 2", dispatcher);

PortSet<string, int> portSet = new PortSet<string, int>();

//vykoná se pouze větev, jejíž zpráva přijde dřív
Arbiter.Activate(taskQueue,
    Arbiter.Choice(portSet, //počet větví dle definice portSet
    zprava => VypisVcetneThreadID(zprava), //příjemce pro string
    zprava => VypisVcetneThreadID(zprava.ToString()) //příjemce pro int
));
portSet.Post("Zkušební text 7-1");
portSet.Post(35);
portSet.Post("Zkušební text 7-2");

Výstup na konzoli:
Zkušební text 7-1
```

Výpis 6: DemoCCR.cs – demoArbiterChoice()

Poznámka 3.3 Existuje více způsobů zápisu arbitru Choice, s některými se setkáme v dalším textu, ostatní nalezne zájemce v dokumentaci k CCR, jež je součástí balíku MRDS, nebo na [14].

- **JoinReceiver**

Často potřebujeme provést nějakou operaci jako reakci na dokončení několika paralelně běžících úloh (viz třeba výše zmíněný příklad s odesláním a archivací mailu). K tomuto účelu slouží třída JoinReceiver. Má několik způsobů použití, z nichž dva si ukážeme na příkladech (výpisy 7 a 8).

```
Dispatcher dispatcher = new Dispatcher(4, "Dispečer");
DispatcherQueue taskQueue = new DispatcherQueue("Fronta", dispatcher);

PortSet<string, int> portSet = new PortSet<string, int>();
//metoda se provede, pouze pokud budou obě zprávy
Arbiter.Activate(taskQueue,
    Arbiter.JoinedReceive(true,portSet.P0,portSet.P1,
    //příjemce - zde anonymní metoda s parametery dle typů obou zpráv
    (zpravaString,zpravaInt) => VypisVcetneThreadID("String: "+zpravaString+", Int: "+
        zpravaInt))
);

portSet.Post("Zkušební text 8-1");
portSet.Post("Zkušební text 8-2");
portSet.Post(35);
portSet.Post("Zkušební text 8-3");
portSet.Post(50);

Výstup na konzoli:
```

```
Vlákno 25 : String : Zkušební text 8−1, Int : 35
Vlákno 27 : String : Zkušební text 8−2, Int : 50
```

Výpis 7: DemoCCR.cs – demoArbiterJoin()

Poznámka 3.4 Zpráva je z portu vytažena pouze pokud jsou k dispozici zprávy na všech požadovaných portech. Proto nehrozí uváznutí (deadlock) v případě, že na podobné kombinace zpráv čeká víc příjemců. Pouze ten, který má k dispozici všechny potřebné zprávy je obsloužen a zprávy jsou z portu odstraněny.

```
VypisVcetneThreadID("Ukázka použití arbitru Join s počtem zpráv"); //výpis včetně
    aktuálního vlákna

Dispatcher dispatcher = new Dispatcher(4, "Dispečer");
DispatcherQueue taskQueue = new DispatcherQueue("Fronta", dispatcher);

Port<string> port = new Port<string>();

int pocetZprav = 10; //kolik pošleme
int prahReakce = 3; //po kolika chceme spustit metodu


Arbiter.Activate(taskQueue,
   // zde odlišný způsob zápisu, nepoužíváme statickou metodu třídy Arbiter
   port.Join(prahReakce,zpravy =>
       { //nactene zpravy jsou predany jako string[]
           foreach (string zprava in zpravy)
               VypisVcetneThreadID(zprava);
       }
   )
);

for (int x = 1; x <= pocetZprav; x++)
    port.Post("Zkušební text 9−" + x);

VypisVcetneThreadID("Konec ukázky − stiskněte klávesu"); //výpis ID aktuálního vlákna
Console.ReadKey();
dispatcher.Dispose(); //ukončení všech běžících vláken

Výstup na konzoli:
Vlákno 09 : Ukázka použití arbitru Join s počtem zpráv
Vlákno 09 : Konec ukázky − stiskněte klávesu
Vlákno 29 : Zkušební text 9−1
Vlákno 29 : Zkušební text 9−2
Vlákno 29 : Zkušební text 9−3
```

Výpis 8: DemoCCR.cs – demoArbiterJoin2()

Příklad z výpisu 8 ukazuje, že můžeme nastavit spuštění obslužné metody také podle počtu přijatých zpráv daného typu. Arbitr je neperzistentní, takže bude metoda zavolána pouze pro první výskyt daného počtu zpráv, ovšem nic nám nebrání v rámci obslužné metody arbitr znovu aktivovat.

- **MultipleItemReceiver**

  Na rozdíl od předchozího příkladu umožňuje tato třída zavolat obslužnou metodu pro určitý počet zpráv v portu, ovšem nezávisle na jejich typu (výpis 9).

```
VypisVcetneThreadID("Ukázka použití arbitru MultipleItemReceiver"); //výpis včetně
    aktuálního vlákna

Dispatcher dispatcher = new Dispatcher(4, "Dispečer");
DispatcherQueue taskQueue = new DispatcherQueue("Fronta", dispatcher);

PortSet<string,int> portSet = new PortSet<string,int>();

int pocetZprav = 10; //kolik pošleme
int prahReakce = 3; //po kolika chceme spustit metodu

Arbiter.Activate(taskQueue,
  portSet.MultipleItemReceive(prahReakce,(zpravyString,zpravyInt) =>
    { //nactene zpravy jsou predany jako string [], int []
      foreach (string zpravaString in zpravyString)
        VypisVcetneThreadID(zpravaString);
      foreach (int zpravaInt in zpravyInt)
        VypisVcetneThreadID(zpravaInt.ToString());
    }
  )
);

for (int x = 1; x <= pocetZprav; x++)
{
  portSet.Post(x);
  portSet.Post("Zkušební text 10-" + x);
}

VypisVcetneThreadID("Konec ukázky - stiskněte klávesu"); //výpis ID aktuálního vlákna
Console.ReadKey();
dispatcher.Dispose(); //ukončení všech běžících vláken

Výstup na konzoli:
Vlákno 09 : Ukázka použití arbitru MultipleItemReceiver
Vlákno 09 : Konec ukázky - stiskněte klávesu
Vlákno 10 : Zkušební text 10-1
Vlákno 10 : 1
Vlákno 10 : 2
```

Výpis 9: DemoCCR.cs – demoArbiterMultipleItemReceiver()

- **Interleave**

  Posledním z arbitrů je třída Interleave. Ta umožňuje specifikovat, které části kódu se mohou vykonávat souběžně různými vlákny a které mají běžet, pouze když neběží jiné vlákno, tedy exkluzivně. Na rozdíl od klasického asynchronního modelu zde programátor pouze určí strategii a vlastní koordinaci vláken už provede na pozadí CCR. Ukázku použití vidíme na výpise 10.

```
VypisVcetneThreadID("Ukázka použití arbitru Interleave"); //výpis včetně aktuálního vlákna

Dispatcher dispatcher = new Dispatcher(4, "Dispečer");
DispatcherQueue taskQueue = new DispatcherQueue("Fronta", dispatcher);

PortSet<string, int, double> portSet = new PortSet<string, int, double>();

Arbiter.Activate(taskQueue,
    Arbiter.Interleave(
        new TeardownReceiverGroup( //po jejím vykonání se ostatní posluchači odregistrují
            // !! Receiver ve skupině Teardown nesmí být perzistentní, ale na chybu
                neupozorní kompilátor, ale až runtime
            Arbiter.Receive<double>(false,portSet,zprava => VypisVcetneThreadID(
                zprava.ToString()))),
        new ExclusiveReceiverGroup( //běží, pouze pokud neběží jiné zpracování zprávy
            Arbiter.Receive<string>(true,portSet,zprava => VypisVcetneThreadID(zprava)
                )),
        new ConcurrentReceiverGroup( //může běžet současně víc zpracování tohoto typu
             zprávy, ale ne jiná
            Arbiter.Receive<int>(true,portSet,zprava => VypisVcetneThreadID(zprava.
                ToString()))
        )
    );
portSet.Post("Zkušební text 11-1");
portSet.Post(1);
portSet.Post("Zkušební text 11-2");
portSet.Post(2);
portSet.Post(3.5);
portSet.Post("Zkušební text 11-3");
portSet.Post(3);
portSet.Post("Zkušební text 11-4");
portSet.Post(4);

//výsledek záleží také na počtu CPU/vláken - kdy se dostane zpráva 3.5 na řadu
VypisVcetneThreadID("Konec ukázky - stiskněte klávesu"); //výpis ID aktuálního vlákna
Console.ReadKey();
dispatcher.Dispose(); //ukončení všech běžících vláken

Výstup na konzoli:
Vlákno 10 : Ukázka použití arbitru  Interleave
Vlákno 12 : Zkušební text 11-1
Vlákno 10 : Konec ukázky - stiskněte klávesu
Vlákno 12 : 1
Vlákno 11 : 2
Vlákno 11 : 3,5
```

Výpis 10: DemoCCR.cs – demoArbiterInterleave()

Poznámka 3.5 Vidíme, že uvnitř arbitru Interleave jsou zapouzdřeny dříve probrané arbitry Receiver. Výše jsme zmínili, že arbitry mohou plánovat jakoukoli třídu implementující rozhraní ITask<>. Protože toto rozhraní implementují i arbitry, lze takto provádět libovolně hluboké zanořování. Tím můžeme velmi elegantně sestavit ce-

lou logiku rozhodování a koordinace zpracování asynchronních požadavků, a to přehledně na jednom místě kódu.

Ve výchozím nastavení mají všechny fronty stejnou prioritu pro všechny zprávy. Potom ale může docházet k tomu, že zpráv v některé frontě bude tolik, že se bude neúměrně zdržovat jejich zpracování (zvlášť pokud nebude k dispozici potřebný počet souběžných vláken). K eliminaci tohoto problému je možné použít řízení výkonu front (throttling). V konstruktoru třídy DispatcherQueue můžeme uvést nepovinný parametr s určením způsobu řízení fronty.

```
VypisVcetneThreadID("Ukázka použití fronty s řízením priority"); //výpis včetně aktuálního vlákna

Dispatcher dispatcher = new Dispatcher(
    8, // potřebujeme více vláken
    "Dispečer"
    );

PortSet<string, int> portSet = new PortSet<string, int>();

int pocetZprav = 8; // kolik pošleme
int maxZprav = 3; //max. velikost fronty

//vytvoření fronty na vlákna
DispatcherQueue taskQueue = new DispatcherQueue("Fronta",dispatcher,
    TaskExecutionPolicy.ConstrainQueueDepthDiscardTasks, //ostatní zahodí
    //TaskExecutionPolicy.ConstrainQueueDepthThrottleExecution, //ostatní čekají
    maxZprav); //kolik zobrazit

Arbiter.Activate(taskQueue,
   Arbiter.Receive<string>(true, portSet, zprava => VypisVcetneThreadID(zprava)),
   Arbiter.Receive<int>(true, portSet, zprava => VypisVcetneThreadID(zprava.ToString()))
);

for (int x = 1; x <= pocetZprav; x++)
{
    //Thread.Sleep(200); //zpomalíme tempo odesílání zpráv
    portSet.Post(x);
    portSet.Post("Zkušební text 12-" + x);
}

VypisVcetneThreadID("Konec ukázky - stiskněte klávesu"); //výpis ID aktuálního vlákna
Console.ReadKey();
dispatcher.Dispose(); //ukončení všech běžících vláken

Výstup na konzoli (při zahazování):
Vlákno 08 : Ukázka použití fronty s řízením priority
Vlákno 11 : 1
Vlákno 11 : Zkušební text 12-2
Vlákno 08 : Konec ukázky - stiskněte klávesu
Vlákno 13 : Zkušební text 12-7
Vlákno 09 : Zkušební text 12-1
Vlákno 15 : Zkušební text 12-8
Vlákno 16 : 8
```

```
Výstup na konzoli (při čekání):
Vlákno 09 : Ukázka použití fronty s řízením priority
Vlákno 12 : 1
Vlákno 11 : Zkušební text 12-1
Vlákno 10 : 2
Vlákno 11 : Zkušební text 12-2
Vlákno 13 : 3
Vlákno 10 : Zkušební text 12-3
Vlákno 10 : 4
Vlákno 10 : Zkušební text 12-4
Vlákno 10 : Zkušební text 12-5
Vlákno 12 : 6
Vlákno 11 : 5
Vlákno 10 : Zkušební text 12-6
Vlákno 10 : 7
Vlákno 10 : Zkušební text 12-7
Vlákno 10 : 8
Vlákno 09 : Konec ukázky - stiskněte klávesu
Vlákno 10 : Zkušební text 12-8

Výstup na konzoli (při zpomalení a zahazování):
Vlákno 09 : Ukázka použití fronty s řízením priority
Vlákno 10 : 1
Vlákno 10 : Zkušební text 12-1
Vlákno 10 : 2
Vlákno 11 : Zkušební text 12-2
Vlákno 10 : 3
Vlákno 11 : Zkušební text 12-3
Vlákno 10 : 4
Vlákno 11 : Zkušební text 12-4
Vlákno 10 : 5
Vlákno 11 : Zkušební text 12-5
Vlákno 11 : 6
Vlákno 10 : Zkušební text 12-6
Vlákno 11 : 7
Vlákno 10 : Zkušební text 12-7
Vlákno 09 : Konec ukázky - stiskněte klávesu
Vlákno 11 : Zkušební text 12-8
Vlákno 10 : 8
```

Výpis 11: DemoCCR.cs – demoDispatcherQueueThrottled()

Na příkladu (výpis 11) vidíme použití této možnosti. Máme na výběr pět metod řízení:

- Unconstrined
  Výchozí nastavení, žádné řízení priority.

- ConstrainQueueDepthDiscardTasks
  Můžeme nastavit maximální délku fronty, zprávy, které se do fronty nevejdou, jsou z portu odstraněny bez zpracování.

- ConstrainQueueDepthThrottleExecution
  Podobně jako v předchozím bodě nastavíme maximální počet zpráv, které jsme schopni zpracovat. Dokud délka aktuální fronty pod tento limit neklesne, je vlákno odesílatele uvedeno do stavu *sleep*.
- ConstrainSchedulingRateDiscardTasks
  Zde nastavíme maximální rychlost příjmu zpráv. Všechny zprávy, které vyžadují rychlejší zpracování, jsou z fronty odstraněny.
- ConstrainSchedulingRateThrottleExecution
  Nastavení stejné jako u předchozího bodu, port nepovolí rychlejší příjem zpráv než stanovený (vlákno odesílatele přejde do stavu *sleep*).

Poznámka 3.6 Obě metody řízení, které nutí odesílatele přejít do režimu *sleep* tedy přenášejí nouzi o systémové prostředky (vlákna=čas CPU) na odesílatele. To je výhodné zejména pokud úkolem odesílatele je pouze generovat zprávy a zpomalení tedy nijak neovlivní ostatní funkce aplikace. Naopak v případě, že se odesílatel spoléhá na to, že co nejrychleji „nasype“ zprávy do portu a pak bude provádět jinou činnost, může být toto nastavení priority příčinou zpomalení jiné části aplikace. Samozřejmě z důvodu řízení procesů běžících pod operačním systémem lze tento postup použít pouze na pozastavení jiného vlákna v rámci stejného procesu OS.

Naznačený model zajišťuje, že se zprávy začnou zpracovávat v pořadí jejich příchod do portu, případně s výše uvedenými možnostmi prioritizace. Pokud ovšem použijeme pro zpracování více typů zpráv jednu frontu (např. při použití PortSet s více typy), jsou všechny zprávy rovnocenné a na přidělení vlákna čekají ve stejné frontě (i throttling považuje všechny zprávy v rámci jedné fronty za sobě rovné). Máme-li tedy nějakou aplikaci, kde se odesílá velké množství zpráv jednoho typu a současně velmi malé množství jiného, nicméně tento druhý typ je pro nás z nějakého důvodu důležitější a potřebujeme jeho okamžité zpracování, můžeme pro zpracování portu příslušného typu vytvořit samostatnou frontu. CCR pak zajišťuje spravedlivé střídání jednotlivých front v přístupu ke zdrojům vláken, takže se druhá fronta dostane k volnému vláknu dřív, než by odpovídalo celkovému pořadí zprávy, kdyby se sečetly oba typy.

3.3.6 Iterator

Dalším problémem, se kterým se setkáme u asynchronních aplikací, je požadavek na vykonávání operací (byť asynchronně) v určitém pořadí. Je tedy nutné, aby *CallBack* metoda nějak zjistila, že už bylo dokončeno zpracování jejího předchůdce a současně nějak informovala svého následníka. Takovémuto vzájemnému řetězení obslužných metod se trefně říká *špagetový model*.

CCR přináší velmi zajímavé řešení tohoto problému využitím iterátorů. Tento prvek je v C# k dispozici od verze 2.0 a umožňuje snadné procházení uživatelsky definovaných kolekcí.[16] Blíže se s nimi může čtenář seznámit v kterékoli novější učebnici jazyka C# (třeba

[16]Typ objektů a způsob jejich procházení lze libovolně definovat, je pouze třeba správně implementovat rozhraní IEnumerable.

[15]). Pro nás je důležité, že se autorům CCR podařilo využít jeho vlastností i na procházení jednotlivých částí kódu. Zjednodušeně řečeno se do iterátoru uloží posloupnost operací určité části kódu a na explicitní žádost CCR iterátor vrátí pozici, na které se má v provádění kódu pokračovat.

Můžeme si to představit jako běžný úsek synchronního kódu, kdy na jednom řádku zavoláme nějakou proceduru, čekáme na její dokončení a teprve poté pokračujeme následujícím řádkem. V části zabývající se principy asynchronního programování (kapitola 3.2) jsme si vysvětlili, že tento postup přímo použít nelze, protože hlavní vlákno aplikace pokračuje ve vykonávání kódu na následujícím řádku ihned (a eliminujeme tedy čekání na provedení předchozího příkazu).

Jak uvidíme na příkladu (výpis 12), podařilo se tvůrcům CCR zachovat strukturu programu známou ze synchronního programování s efektivitou asynchronního. Je třeba si uvědomit jeden podstatný rozdíl, který není na první pohled patrný. Zatímco u synchronního modelu vlákno čekalo na dokončení příkazu, zde se vlákno (jako systémový prostředek) ihned vrátí a je k dispozici pro jinou část aplikace. Teprve po dokončení požadované operace si CCR znovu vyžádá volné vlákno a pokračuje dalším příkazem. Ve výsledku je tedy aplikace plně asynchronní, přestože při psaní kódu postupujeme jako bychom psali kód synchronní.

```
public void demoIterator()
{
    Console.Clear();
    VypisVcetneThreadID("Ukázka použití iterátoru"); //výpis včetně aktuálního vlákna

    Dispatcher dispatcher = new Dispatcher(4,"Dispečer");

    //vytvoření fronty na vlákna
    DispatcherQueue taskQueue = new DispatcherQueue("Fronta", dispatcher);

    Arbiter.Activate(taskQueue,
        Arbiter.FromIteratorHandler(enumerator) //vlastní spojení iterátoru s frontou
    );

    for (int x = 1; x <= 5; x++)
    {
        Thread.Sleep(50); //trošku zpomalíme cyklus, aby neproběhl celý najednou
        VypisVcetneThreadID("hlavní vlákno" + x);
    }

    VypisVcetneThreadID("Konec ukázky - stiskněte klávesu"); //výpis ID aktuálního vlákna
    Console.ReadKey();
    dispatcher.Dispose(); //ukončení všech běžících vláken
}

IEnumerator<ITask> enumerator()
{
    int pocetZprav = 10; //kolik pošleme

    Port<string> port= new Port<string>();
```

```
    for (int x=1;x<=pocetZprav;x++)
    {
        if (x==5)
            yield break; //ukončí iterátor =další příkazy už nebudou provedeny

        VypisVcetneThreadID(x+". vstup do smyčky");
        port.Post("Zkušební text 13-"+x);

        // zde chceme počkat až do zpracování zprávy
        yield return port.Receive( //jiný způsob vytvoření neperz. arbitra Receiver
            zprava =>
            {
                Thread.Sleep(100);// simulace delšího zpracování zprávy
                VypisVcetneThreadID(zprava);
            }
        );
        VypisVcetneThreadID(x + ". konec smyčky");// sem se vracíme po zpracování
    }
}
```

```
Výstup na konzoli:
Vlákno 10 : Ukázka použití iterátoru
Vlákno 11 : 1. vstup do smyčky
Vlákno 10 : hlavní vlákno1
Vlákno 10 : hlavní vlákno2
Vlákno 11 : Zkušební text 13-1
Vlákno 11 : 1. konec smyčky
Vlákno 11 : 2. vstup do smyčky
Vlákno 10 : hlavní vlákno3
Vlákno 11 : Zkušební text 13-2
Vlákno 11 : 2. konec smyčky
Vlákno 11 : 3. vstup do smyčky
Vlákno 10 : hlavní vlákno4
Vlákno 10 : hlavní vlákno5
Vlákno 10 : Konec ukázky - stiskněte klávesu
Vlákno 11 : Zkušební text 13-3
Vlákno 11 : 3. konec smyčky
Vlákno 11 : 4. vstup do smyčky
Vlákno 11 : Zkušební text 13-4
Vlákno 11 : 4. konec smyčky
```

Výpis 12: DemoCCR.cs – demoIterator()

Vidíme, že v průběhu obsluhy zpráv iterátoru stále běží hlavní vlákno, takže není zpracováním brzděno, a přitom máme zajištěno, že jsou zprávy vykonávány v přesně určeném pořadí.[17]

Poznámka 3.7 CCR předá řízení další operaci iterátoru pouze v případě, že je první operace dokončena. Musíme si ovšem uvědomit, že ve výše uvedeném příkladu je touto operací arbitr Receiver, který zajišťuje zpracování zprávy na daném portu. Protože je

[17] Na místě testovacího řetězce si pochopitelně spíše představme nějakou užitečnou operaci, jako je například vkládání záznamů do databáze, kde rovněž většinou požadujeme přesné pořadí kroků.

neperzistentní, je zrušen ihned po příjmu první zprávy a jeho úkol je tím naplněn. Kdybychom ho však definovali jako perzistentní, stále by do fronty vkládal dvojice zpráva–příjemce a požadoval jejich provedení vláknem. Byl by tedy stále aktivní a řízení by se nevrátilo.

Další příklady použití iterátoru nalezne zájemce v dokumentaci CCR [14], zde je pro nedostatek místa nebudeme uvádět. Na předchozím příkladu (výpis 12) je přesto naznačeno, že lze na základě nějaké podmínky provádění těla iterátoru také ukončit. Vzpomeneme-li si na výše představené druhy arbitrů, zřejmě právě zde nalezneme uplatnění Choice a Interleave.

3.3.7 Řešení chyb

V několika z předchozích příkladů bylo demonstrováno, jak lze v CCR identifikovat a řešit situace, kdy v některé části kódu dojde k chybě. V této souvislosti lze nejlépe využít arbitry Choice a Interleave.

CCR však umožňuje i pokročilejší techniku řízení kódu při výskytu chyb, kdy zpracování chyb nějakého většího celku kódu soustředíme na jednom místě, kam všechna hlášení o chybách dopravíme. V CCR tuto úlohu plní třída Causality.

Instance této třídy je vždy přiřazena konkrétnímu vláknu, ve kterém se deklarace nachází. Všechny porty, které v kontextu tohoto vlákna vytvoříme, si pak kauzalitu nesou s sebou a přenáší se i na vlákna, která zprávy těchto portů zpracovávají. Takto máme zajištěno, že pokud na jednom místě kauzalitu definujeme, dostaneme se ke všem výjimkám vzniklým na této a všech nižších úrovních.

Na následujícím příkladu (výpis 13) uvidíme, jak lze jednoduše každému dispečerovi přiřadit instanci Causality, čímž zajistíme, že všechny výjimky, které se objeví ve vláknech spravovaných daným dispečerem, budou ošetřeny na jednom místě.

```
VypisVcetneThreadID("Ukázka použití třídy Causality pro zachytávání výjimek"); //výpis včetně
    aktuálního vlákna

Dispatcher dispatcher = new Dispatcher(4,"Dispečer");

Port<string> port = new Port<string>();

DispatcherQueue taskQueue = new DispatcherQueue("Fronta", dispatcher);

int pocetZprav=4; //kolik pošleme
int ukazatel = 0;
string[] poleStringu=new string[pocetZprav-2]; //o 1 kratší, aby vyskočila výjimka

// arbitr pro zpracování běžných zpráv
Arbiter.Activate(taskQueue,
   Arbiter.Receive(true, port, zprava =>
      {
         poleStringu[ukazatel++] = zprava; //zde čekám na výjimku indexu pole
         VypisVcetneThreadID(zprava); //kontrolní výpis po úspěšném uložení
      }
   )
```

```
);

Port<Exception> portChybovek=new Port<Exception>(); //sem se budou vkládat výjimky
Causality kauzalita = new Causality("Kauzalita", portChybovek);

//musíme vytvořit i arbitra pro zpracování chybovek, zde používá stejnou frontu
Arbiter.Activate(taskQueue,
   Arbiter.Receive(true, portChybovek, zprava => VypisVcetneThreadID(zprava.Message+": " +
       ukazatel)));

Dispatcher.AddCausality(kauzalita); //přiřazení arbitra

for (int x = 1; x <= pocetZprav; x++) //odešleme dávkově několik zpráv
    port.Post("Zkušební text 14-" + x);

VypisVcetneThreadID("Konec ukázky - stiskněte klávesu"); //výpis ID aktuálního vlákna
Console.ReadKey();
dispatcher.Dispose(); //ukončení všech běžících vláken

Výstup na konzoli:
Vlákno 10 : Ukázka použití třídy Causality pro zachytávání výjimek
Vlákno 10 : Konec ukázky - stiskněte klávesu
Vlákno 12 : Zkušební text 14-1
Vlákno 14 : Zkušební text 14-4
Vlákno 11 : Index je mimo hranice pole.: 4
Vlákno 13 : Index je mimo hranice pole.: 4
```

Výpis 13: DemoCCR.cs – demoCausality()

Poznámka 3.8 Všimněme si, že ačkoli mají anonymní metody v těle arbitrů přístup k lokálním proměnným hlavního programu (což je jedna z jejich výhod), zde dochází k nekoordinovanému přístupu více vláken k jedné proměnné (ukazatel) najednou. Proto byly do pole uloženy nikoli první dva odeslané řetězce, ale první a poslední. Vlákno zpracovávající poslední řetězec zřejmě přečetlo proměnnou dříve než zbývající dvě. Vidíme ideálního kandidáta na použití iterátoru.

Kauzality můžeme také libovolně vnořovat, případně z vnořených úrovní naopak výjimky přeposílat úrovním vyšším, třeba dle jejich typu, a zajistit tak, že každá výjimka bude obsloužena, pokud možno vždy na té úrovni, na které víme, že na ni dokážeme vhodně reagovat.

Stejně tak platí, že nemusíme strukturu kauzalit měnit ani v případě spojování portů např. při zpracování arbitrem Join, který v případě výskytu výjimky v kontextu svého vlákna ji přepošle kauzalitám všech zúčastněných portů.

3.3.8 Shrnutí

Závěrem si uveďme některé další specifické vlastnosti CCR:

- Důležitým rozdílem proti mechanismu *událostí*, jak jej známe z C#, je například to, že můžeme zprávy do portu odeslat dříve, než provedeme registraci příjemce. Zprávy v portu tak čekají, až je bude mít kdo zpracovat.

- Běžný přístup k programování asynchronních aplikací pod .NET vyžaduje pevné svázání *CallBack* metody s určitou operací. CCR umožňuje toto chování dynamicky měnit směrováním zpráv na různé porty, jejich větvením, slučováním, podmínkami apod.
- Na rozdíl od mechanismu událostí zaručuje CCR afinitu zpracování zpráv vůči zdrojům vláken určeným při registraci dispečera. Příjemce tak vždy běží ve vlákně ze zdroje určeného příslušným dispečerem.
- Prostřednictvím řízení výkonu (throttling) lze zajistit snížení náročnosti aplikace zpracováním pouze některých zpráv.
- Správa vláken funguje na podobném principu jako ThreadPool s QueueUserWorkItem (a je s ním prostřednictvím třídy Dispatcher zpětně kompatibilní), nicméně poskytuje daleko pružnější organizaci poskytování a vracení vláken.
- Systém zachytávání výjimek v CCR nutí programátora je také konzistentně ošetřit. Pokud se totiž rozhodne např. použít arbitra Interleave, musí nutně vytvořit obslužnou metodu pro větev Teardown.
- CCR svou podstatou plně asynchronního modelu umožňuje situaci, kdy hlavní procedura (případně vlákno celé aplikace) skončí a přesto bude dál probíhat zpracování zpráv v jiných vláknech. Proto je někdy nutné hlavní kód aplikace uměle udržovat při životě (keep it busy), přestože nemá nic užitečného na práci, případně před ukončením očekávaným aplikace zrušit všechny dispečery (zavolat Dispose). Lze nastavit i jiné chování, kdy se společně s koncem aplikace ukončí i všechna jí vyvolaná vlákna.[18]
- Odesílání zpráv do portu, který ještě nemá připojené své příjemce, je na jedné straně pro programátora velmi pohodlné, na druhé straně je však třeba mít na paměti, že instance portu běží pod vláknem, které jej vytvořilo a až do jeho ukončení tedy *Garbage Collector* nebude zprávy na něj zaslané rušit. To může v případě velmi zatíženého neobsluhovaného portu vést ke zbytečným ztrátám paměti. Podobné následky může mít i příliš pomalé zpracování zpráv ve srovnání s rychlostí jejich generování. Zde je na místě použití některé z metod řízení výkonu (throttling).

[18] Ještě před vytvořením prvního vlákna je třeba nastavit Dispatcher.UseBackThreads = **true**.

4 Decentralized Software Services (DSS)

4.1 Vysvětlení pojmu

DSS je běhové prostředí, které je postaveno nad CCR a umožňuje tvorbu a běh odlehčených samostatných programových modulů, které spolu komunikují výhradně zprávami, mohou běžet lokálně na stejném stroji nebo být distribuovány v celé síti.

Stejně jako v případě CCR, i zde měli autoři na mysli zejména zjednodušení návrhu aplikací, jejich snadnou spolupráci a nasazení v rámci libovolné hardwarové platformy.

Z pohledu tohoto textu, primárně zaměřeného na jazyk VPL, je právě DSS tím stupněm, který leží bezprostředně pod programovými bloky VPL a zprostředkovává jim dříve popsané funkce a vlastnosti CCR.

Poznámka 4.1 Níže prezentovaný popis DSS nebude ani zdaleka úplný, je zaměřen zejména na styčné body mezi DSS, CCR a VPL. Podrobné vysvětlení všech pojmů a možností prostředí DSS může zájemce najít například v dokumentaci dodané společně s MRDS, případně na [16].

4.2 Struktura služby

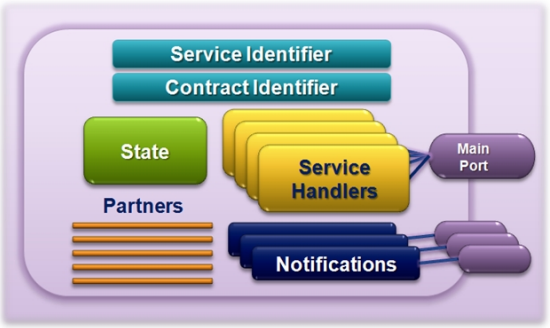
Základní jednotkou aplikace psané pro DSS je služba. Její struktura je schematicky znázorněna na obrázku 3.

Stručně si vysvětlíme označení bloků, které na obrázku vidíme:

- **Service Identifier**
  Unikátní název konkrétní instance služby v rámci daného *DSS uzlu*.[19]
- **Contract Identifier**
  Adresa popisu kontraktu. Kontrakt (Contract) slouží k popisu služby (jejího rozhraní a chování) a z něho jiné služby čerpají informace o způsobu komunikace s ní. Jde o prostý XML soubor a lze jej pohodlně editovat.
- **State**
  Stav služby – popisuje aktuální stav služby z hlediska plnění její funkce. Jde o informaci, kterou o sobě služba poskytuje navenek svým partnerům. V zásadě je to soubor proměnných, ve kterých si služba uchovává informace.
- **Partners**
  Seznam služeb, na nichž daná služba závisí. Tyto služby se pokusí běhové prostředí před startem vlastní služby také lokalizovat, případně spustit.[20]
- **Main Port**
  Jde o jediný vstupní bod služby, jímž přijímá požadavky svých partnerů. Se znalostmi CCR si již umíme představit, že jde vlastně o CCR port, na který je uvnitř služby navázáno zpracování došlých zpráv.

[19] Samostatný uzel sítě zajišťující běh služeb na svěřeném HW.

[20] Lze nastavit, zda se služba může spustit i po neúspěšném pokusu o spuštění svých partnerů.



Obrázek 3: Struktura DSS služby [17]

- **Service Handlers**
  Jednotlivé typy zpráv, jež podporuje *Main Port*, odpovídají funkcím služby. Každý typ zprávy potom potřebuje obslužnou proceduru (handler), která zprávu zpracuje. Množina těchto procedur pak tvoří Service Handlers.

- **Notifications**
  Mimo odpovědi na požadavky partnerů může služba také poskytovat nevyžádané zprávy na základě změny svého vnitřního stavu. Tyto zprávy se nazývají notifikace (Notifications).[21]

Aby služba své rozhraní zpřístupnila jiným službám, používá takzvanou *dll proxy*.

Ve vlastním kódu můžeme specifikovat, které metody a vlastnosti zveřejníme prostřednictvím označení atributy DataContract Attribute , DataMemberAttribute a DataMember Constructor Attribute . Více se o nich dozvíme z konkrétních příkladů v kapitole 4.8. V důsledku pak hrají klíčovou roli ve stavebních prvcích VPL, kde určují, jaké vlastnosti v editoru uvidíme a můžeme tedy využívat.

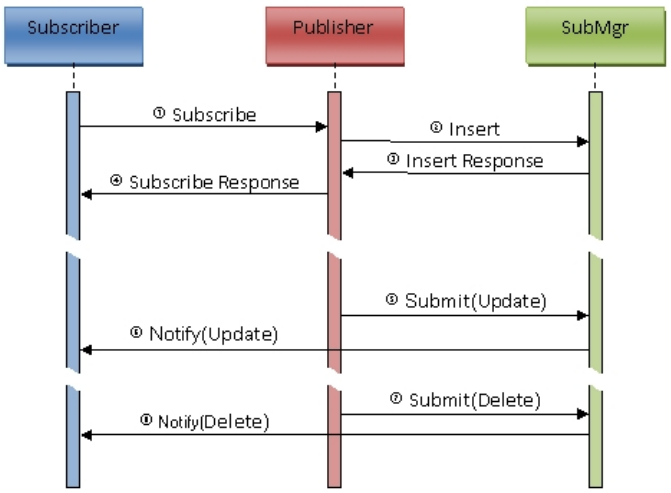
4.3 Komunikace

Celá architektura DSS je založena na komunikaci přes *DSSP*.[22]. Jde o poměrně jednoduchý textový protokol na základě *HTTP*, kterým si služby vyměňují tzv. *SOAP*[23] zprávy. Proto i všechny služby (jejich společný rodič) mají schopnost přijímat požadavky *HTTP Get* a odpovídat na ně. Výhodou potom je, že se službami můžeme komunikovat i běžným webovým prohlížečem. Do adresového řádku pouze uvedeme identifikátor služby a v okně prohlížeče bychom měli vidět její odpověď. Podobu odpovědi určuje vlastní implementace zpracování zprávy *HTTP Get* službou. Většinou uvidíme XML soubor, ale pro některé služby se přímo nabízí jejich stav prostřednictvím *XSLT* transformací

[21] Možná přesnějšími českými ekvivalenty by byly oznámení nebo upozornění, ale raději se budeme držet označení bližšího originálu, zejména z důvodu přehlednosti kódu.

[22] Specifikaci protokolu viz [18]

[23] Simple Object Access Protocol – jde o textový protokol na bázi XML.



Obrázek 4: Princip registrace notifikací [16]

prezentovat jako HTML stránku. Také tuto možnost DSS podporuje a jednoduchý příklad takové transformace lze nalézt v dokumentaci MRDS.

4.4 Notifikace

O tom, že spolu služby komunikují také prostřednictvím notifikací, jsme se už zmínili v úvodu kapitoly. Aby služba věděla, komu má notifikaci o změně svého stavu odeslat, obrací se na systémovou službu *DSS Subscriber Manager*.[24] Ta spravuje její seznamy registrovaných příjemců jednotlivých notifikací. Na obrázku 4 vidíme proces výměny zpráv mezi jednotlivými stranami transakce při registraci nového zájemce o notifikaci. Odběratel (Subscriber) se nejprve obrátí na vydavatele (Publisher) se žádostí o přihlášení k odběru notifikací. Vydavatel pak požádá svého *Subscriber Managera* o založení nového odběratele a o výsledku odběratele informuje. Při změně vnitřního stavu pošle služba notifikaci svému manažerovi a ten všem přihlášeným odběratelům zprávu přepošle.[25]

4.5 Uživatelské rozhraní

Z hlediska DSS může i uživatelské rozhraní aplikace a jeho jednotlivé prvky představovat samostatné služby, které jsou schopny reagovat na vstupní požadavky a naopak poskytovat informace o svém stavu. K MRDS, resp. knihovně DSS, jsou přiloženy vzorové ukázky takovýchto rozhraní (včetně webových) a je možné je libovolně rozšiřovat. Propojení uživatelského rozhraní na bázi *WPF* prostřednictvím DSS je také námětem jednoho ze vzorových příkladů v dokumentaci MRDS (pod názvem *WPF Text To Speech UI*).

[24] Většinou si každá služba vytváří vlastní instanci manažera
[25] Jde o obdobu modelu událostí z C#.

4.6 Nástroje

Součástí balíku DSS v MRDS je i řada nástrojů spustitelných z příkazové řádky, které mohou pomoci s vývojem nebo laděním DSS služeb. Zejména užitečné jsou utility pro vyhledání služeb, přečtení jejich popisu, konfigurace jejich manifestů apod.

4.7 Generické služby

O nich se dozvíme více v samotné sekci VPL (kapitola 7.2), ale jde v podstatě o univerzální služby, které poskytují nějaké obecně definované funkce a které můžeme specializovat pro konkrétní nasazení. Ve VPL jsou takto třeba definovány obecné analogové senzory a na výrobci HW je pak implementace komunikace s reálným HW a doplnění o specifické názvosloví, kalibraci hodnot apod.

4.8 Vytvoření služby DSS

Bohužel,[26] tímto tématem se v originální dokumentaci Microsoftu zabývá celý dvanáctistránkový tutoriál a není možné zde postihnout všechny možnosti. Proto si ukážeme postup opačný než u CCR, kde jsme postupovali od základních bloků k celku. Tady si představíme zdrojový kód hotové DSS služby a v krátkosti si popíšeme jednotlivé jeho bloky, abychom získali základní představu o jeho organizaci.

Nyní tedy již k samotné službě. Z úvodu kapitoly 4 již víme, že jednotlivé služby spolu komunikují výhradně pomocí zpráv a také víme, že každá služba obsahuje jeden hlavní port, jeden port určený pro notifikace a pak redirektor pro komunikaci s jinými službami. Protože všechny služby založené na DSS musí povinně implementovat některé základní typy zpráv[27], vytvoří nám pro nový projekt typu DSS služby Visual Studio rovnou takový prototyp služby, který již tuto podmínku splňuje. Odpadá nám tedy starost s definicí jednotlivých portů a prvním úkolem pak je definovat nějakou vlastní zprávu. Teoreticky bychom mohli postupovat přesně dle principů CCR a zprávy a jejich příjemce definovat přímo, ale DSS přináší vyšší úroveň abstrakce a detaily implementace na nižší úrovni provádí samo. Toho můžeme s výhodou využít a zprávy definovat dle připravených vzorů.

Každá služba vyvíjená v prostředí Visual Studia je tvořena dvěma základními soubory se zdrojovým kódem. První z nich vidíme na výpise 14.

Protože z úsporných důvodů byly z výpisů odstraněny automaticky generované komentáře jednotlivých metod, je vhodné v případě pochybností o správném porozumění jejich účelu prostudovat příklady na přiloženém CD, kde jsou výpisy včetně oněch komentářů.

```
1  {
2      public sealed class Contract
3      {
```

[26] Ale naštěstí pro potenciální autory DSS služeb

[27] Jejich úplný výčet najdeme v dokumentaci, nicméně jednou z nich je již dříve zmíněný požadavek *HTTP Get*.

```
4           [DataMember]
5           public const string Identifier = "http :// schemas.tempuri.org/2010/03/demodss_service.
                html";
6       }
7
8       [DataContract]
9       public class DemoDSS_ServiceState
10      {
11          private int _pocitadlo = 0;
12          private string _text = "";
13          private string _informace = "";
14
15          [DataMember] //bude serializován (součástí SOAP zprávy)
16          public string Informace
17          {
18              get { return _informace; }
19              set { _informace = value; }
20          }
21
22          //nebude součástí SOAP zprávy – neuvidíme zvenku
23          public int Pocitadlo
24          {
25              get { return _pocitadlo; }
26              set { _pocitadlo = value; }
27          }
28
29          [DataMember] //bude serializován (součástí SOAP zprávy)
30          public string Text
31          {
32              get { return _text; }
33              set { _text = value; }
34          }
35
36      }
37
38      [ServicePort] //PortSet určuje, které typy zpráv budeme akceptovat, musíme sem přidat námi
            vytvořené
39          //jde pouze o pojmenovaný soubor operací, které podporuje naše služba
40      public class DemoDSS_ServiceOperations : PortSet<DsspDefaultLookup, DsspDefaultDrop,
            Get, Subscribe, HttpGet, TextNaVelka, TextNaMala>
41      {
42      }
43
44          public class Get : Get<GetRequestType, PortSet<DemoDSS_ServiceState, Fault>>
45      {
46
47          public Get()
48          {
49          }
50
51          public Get(GetRequestType body)
52              : base(body)
53          {
54          }
```

```
55
56          /// <summary>
57          /// Creates a new instance of Get
58          /// </summary>
59          /// <param name="body">the request message body</param>
60          /// <param name="responsePort">the response port for the request</param>
61          public Get(GetRequestType body, PortSet<DemoDSS_ServiceState, Fault> responsePort)
62              : base(body, responsePort)
63          {
64          }
65      }
66
67      public class Subscribe : Subscribe<SubscribeRequestType, PortSet<
            SubscribeResponseType, Fault>>
68      {
69          public Subscribe()
70          {
71          }
72
73          public Subscribe(SubscribeRequestType body)
74              : base(body)
75          {
76          }
77
78          public Subscribe(SubscribeRequestType body, PortSet<SubscribeResponseType, Fault>
                responsePort)
79              : base(body, responsePort)
80          {
81          }
82      }
83
84      //################################################################################
85      // Vytvoříme nový typ zprávy TextNaVelka s obálkou na zasílaný požadavek
86      //################################################################################
87      [DataMemberConstructor]
88      public class TextNaVelka : Replace<TextNaVelkaRequest, PortSet<DemoDSS_ServiceState,
            Fault>>
89      {
90          public TextNaVelka()
91          {
92          }
93
94          public TextNaVelka(string text)
95              //  jeden z konstruktorů Replace<TBody,TResponse> je Replace(TBody)
96              : base(new TextNaVelkaRequest(text)) //název zprávy a tělo zprávy
97          {
98          }
99      }
100
101     [DataMemberConstructor]
102     [DataContract]
103     public class TextNaVelkaRequest
104     {
105         private string _text ;
```

```
106
107         public TextNaVelkaRequest()
108         {
109         }
110
111         public TextNaVelkaRequest(string text)
112         {
113             _text = text;
114         }
115
116         [DataMember] //v proxy bude pouze konstruktor, který obsahuje tento parametr
117         public string Text
118         {
119             get { return _text; }
120             set { _text = value; }
121         }
122     }
123
124     //#############################################################################
125     //Vytvoříme nový typ zprávy TextNaMala s obálkami na zasílaný požadavek i návratovou
            hodnotu
126     //#############################################################################
127     [DataMemberConstructor]
128     public class TextNaMala : DsspOperation<TextNaMalaRequest, PortSet<
            TextNaMalaResponse, Fault>>
129     {
130         public TextNaMala()
131             : base(DsspActions.QueryRequest) //musíme specifikovat typ akce
132         {
133         }
134
135         public TextNaMala(string text)
136             // univezální, proto zde proti Replace musíme specifikovat název typu zprávy
137             : base(DsspActions.QueryRequest, new TextNaMalaRequest(text)) //název typu
                    zprávy a tělo zprávy
138         {
139         }
140     }
141
142     //vytvoříme vlastní obálku na zasílanou hodnotu
143     [DataMemberConstructor]
144     [DataContract]
145     public class TextNaMalaRequest
146     {
147         private string _text;
148
149         public TextNaMalaRequest()
150         {
151         }
152
153         public TextNaMalaRequest(string text)
154         {
155             _text = text;
156         }
```

```
157             [DataMember] //v proxy bude pouze konstruktor, ktery obsahuje tento parametr
158             public string Text
159             {
160                 get { return _text; }
161                 set { _text = value; }
162             }
163         }
164
165         //vytvoříme vlastní obálku na návratovou hodnotu
166         [DataMemberConstructor]
167         [DataContract]
168         public class TextNaMalaResponse
169         {
170             private string _text;
171
172             public TextNaMalaResponse()
173             {
174             }
175
176             public TextNaMalaResponse(string text)
177             {
178                 _text = text;
179             }
180
181             [DataMember] //v proxy bude pouze konstruktor, ktery obsahuje tento parametr
182             public string Text
183             {
184                 get { return _text; }
185                 set { _text = value; }
186             }
187         }
188     }
```

Výpis 14: DemoDSS_Service.cs

Na úvod bychom si měli říci, co je vlastně úkolem služby z předchozího výpisu. Vzhledem ke způsobu prezentace výsledků služby na tištěném médiu nebylo možné zvolit žádnou složitější službu s multimediálním obsahem, případně se vztahem k ovládání robotů. Pro jednoduchou demonstraci tedy musí postačit prostá funkce převodu zadaného textu na velká resp. malá písmena. Každá z těchto funkcí je implementována poněkud odlišným způsobem, který nám usnadní pochopit některé hlavní principy vytváření služeb.

Konkrétně tedy budeme požadovat, aby služba provedla konverzi námi zaslaného textu řetězce typu String na velká resp. malá písmena. Současně požadujeme, aby při požadavku na konverzi na velká písmena uložila tento text jako součást svého vnitřního stavu a celý obsah svého stavu nám poslala v odpovědi. V případě konverze na malá písmena pošle pouze zkonvertovaný text a svůj stav nemění.[28] V případě obou konverzí

[28] Pro účely ladění a prezentace výstupů je součástí vnitřního stavu i textová vlastnost Informace, která se tak technicky mění i při konverzi na malá písmena.

pak bude jejich celkový počet evidovat jednoduchým počítadlem, které je rovněž součástí stavu služby.

Abychom situaci dále nekomplikovali implementací nějakého uživatelského rozhraní, použijeme s výhodou standardně definovanou odpověď služby na HTTP požadavek Get. Ta vrací ve formě XML souboru obsah svého stavu a můžeme tak velmi snadno ke službě přistupovat z webového prohlížeče.

Vytvoříme si tedy dvě téměř identické služby, z nichž první bude sloužit jako překladatel (bude vracet zkonvertované texty) a druhá bude na základě jednotlivých požadavků Get od první služby tyto konverze požadovat, ukládat je do svého stavu a prezentovat jako XML ve webovém prohlížeči.

Pro některé čtenáře bude zřejmě nepříjemným překvapením, že pro tak triviální operaci bylo třeba napsat několik desítek řádek kódu. Abychom tedy rozptýlili zbytečné obavy, řekněme si, že vlastní výkonné jádro námi požadovaných funkcí se nachází na řádcích č. 42 až 44 výpisu 15. Ostatní kód je z větší části vygenerován Visual Studiem už při vytvoření nového projektu typu DSS služba a z menší části se jedná o velmi jednoduché „obálky" základních tříd a metod, které se vesměs budou ve všech nově vytvářených službách opakovat, snad s malými obměnami.

Je tedy na místě předpokládat, že pro složitější projekty si jejich autor vytvoří vlastní šablony a prefabrikované vzory[29] a režie tak bude minimální.

Nyní, když víme, co služba provádí, podívejme se detailněji na výpis jejího kódu.

Na řádku 4 vidíme popis kontraktu. Jde o unikátní adresu, na kterou se pak budou obracet jiné služby. Protože v našem případě všechny služby poběží lokálně na jednom PC, můžeme ponechat výchozí adresu generovanou VS.

Na řádcích 9-36 je pak popis stavu služby. Jde o informace, které si o sobě služba uchovává a které případně poskytuje směrem ven. K tomu musí mít privátní proměnné deklarovány jako vlastnosti a v případě jejich zasílání *DSSP* zprávami také musí mít atribut DataMember.

Na řádku 40 se deklaruje typ zpráv, které bude služba ochotna přijmout na svém hlavním portu. Sem musíme přidat všechny zprávy, které si sami nadefinujeme. Proto na konci seznamu vidíme typy TextNaVelka, TextNaMala.

Na dalších několika řádcích jsou pak jednotlivé třídy těchto zpráv, včetně svých vstupních a výstupních formátů a konstruktorů. My přeskočíme na řádek 84, kde začíná námi definovaná třída nové zprávy TextNaVelka.

DSS nabízí několik předdefinovaných typů zpráv, které odpovídají požadavkům protokolu *DSSP*, proto můžeme jeden z nich využít. Naši zprávu tedy zdědíme od třídy Replace, která slouží k zaslání nového stavu jiné třídě. Ta pak tímto stavem nahradí svůj vlastní a o provedení informuje buď potvrzující, nebo chybovou zprávou.

Na řádku 88 tak říkáme, že naše služba bude posílat data požadavku zabalená ve třídě TextNaVelkaRequest, což se v terminologii DSS označuje jako tělo (Body) zprávy a odpověď

[29] Třeba ve formě generických služeb.

bude přijímat na portech podporujících příjem celého stavu služby (DemoDSS_ServiceState – viz řádky 9-36) a standardních chyb Fault.[30]

V konstruktoru třídy (musíme uvést i defaultní prázdný) pak vytvoříme novou instanci žádosti, do které vložíme svá data (zde jeden řetězec).

Na řádcích 103-121 je vlastní třída tohoto požadavku, včetně konstruktorů a vlastností. Důležité je u vlastností, které mají být nastaveny v konstruktoru, uvést atribut DataMember, čímž si říkáme o to, aby pro tuto třídu zpřístupnila proxy pouze ten konstruktor, který danou vlastnost obsahuje jako svůj parametr.

Velmi podobně navrhneme i zprávu pro konverzi na malá písmena. Vidíme pouze, že již nedědíme od třídy Replace, ale od DsspOperation. Jde o obecnější třídu (je to rodič Replace), kterou můžeme použít pro volnější definici vlastních zpráv.

Naše třída vypadá podobně jako v předchozím případě, pouze návratovou zprávu již netvoří celý stav služby, ale pouze další „obálka“,ve které se přepraví požadovaná data (zde zkonvertovaný řetězec).

Protože dědíme od obecné třídy DsspOperation, nesmíme v konstruktoru zapomenout nastavit typ akce, kterou budeme provádět. Specializované třídy, jako třeba právě Replace nebo Get již tento typ mají nastavený ve vlastních konstruktorech.

```
1  namespace DemoDSS_Service
2  {
3      // atributy slouží k popisu služby
4      [Contract(Contract.Identifier)]
5      [DisplayName("DemoDSS_Service")]
6      [Description("DemoDSS_Service_service_(no_description_provided)")]
7      class DemoDSS_Service : DsspServiceBase
8      {
9          [ServiceState]
10         DemoDSS_ServiceState _state = new DemoDSS_ServiceState();
11
12         [ServicePort("/DemoDSS_Service", AllowMultipleInstances = true)]
13         DemoDSS_ServiceOperations _mainPort = new DemoDSS_ServiceOperations();
14
15         [SubscriptionManagerPartner]
16         submgr.SubscriptionManagerPort _submgrPort = new submgr.SubscriptionManagerPort();
17
18         public DemoDSS_Service(DsspServiceCreationPort creationPort)
19             : base(creationPort)
20         {
21         }
22
23         protected override void Start()
24         {
25             // Add service specific initialization here
26             base.Start();
27         }
28
29         [ServiceHandler]
```

[30] Zde vidíme jasně návaznost na CCR, kdy je definován PortSet a zpráva se správně přiřadí do odpovídajícího portu dle svého typu.

```
30          public void SubscribeHandler(Subscribe subscribe)
31          {
32              SubscribeHelper(_submgrPort, subscribe.Body, subscribe.ResponsePort);
33          }
34
35          //######################################################################
36          //Musíme vytvořit příjemce schopné reagovat na zprávy našeho typu
37          //######################################################################
38
39          [ServiceHandler(ServiceHandlerBehavior.Exclusive)] //budeme přistupovat k počítadlu,
                takže exkluzivní běh vlákna
40          public IEnumerator<ITask> TextNaVelkaHandler(TextNaVelka zprava) //používáme iterátor
                , abychom neblokovali hlavní vlákno
41          {
42              _state.Pocitadlo++; //pouze evidujeme počet vykonaných překladů
43              _state.Text = zprava.Body.Text.ToUpper();
44              _state.Informace = String.Format("Dosud vykonáno: {0}, poslední: {1}",_state.
                    Pocitadlo,zprava.Body.Text);
45              zprava.ResponsePort.Post(_state); //vrať pozitivní (defaultní) odpověď
46              yield break; //čekej na další pokyn ke zpracování zprávy (vrať vlákno)
47          }
48
49          [ServiceHandler(ServiceHandlerBehavior.Exclusive)] //budeme přistupovat k počítadlu,
                takže exkluzivní běh vlákna
50          public IEnumerator<ITask> TextNaMalaHandler(TextNaMala zprava) //používáme iterátor,
                abychom neblokovali hlavní vlákno
51          {
52              // Only update the state here because this is an Exclusive handler
53              _state.Pocitadlo++;
54              String odpoved = zprava.Body.Text.ToLower();
55              _state.Informace = String.Format("Dosud vykonáno: {0}, poslední: {1}", _state.
                    Pocitadlo, odpoved);
56              zprava.ResponsePort.Post(new TextNaMalaResponse(odpoved)); //vrať pozitivní (
                    defaultní) odpověď
57              yield break; //čekej na další pokyn ke zpracování zprávy (vrať vlákno)
58          }
59      }
60  }
```

Výpis 15: DemoDSS_Service.cs

Na výpise 15 druhého zdrojového souboru vidíme hned po hlavičce (dědíme od třídy DsspServiceBase, čímž máme mimo jiné zajištěnu obsluhu standardních zpráv) vytvoření instance vlastního stavu služby (řádek 10), hlavního portu pro příjem zpráv (13) a portu pro komunikaci s manažerem spravujícím přihlášené odběratele notifikací.

Následuje konstruktor (řádek 18) a inicializační procedura, která se spustí po startu služby. Sem můžeme přidat operace, které požadujeme v této fázi provést.

Dále vidíme iterátor, který slouží pro příjem zprávy typu Subscribe (řádek 30), jeho obsah pro nás není momentálně podstatný, je ale zřejmé, že přeposílá požadavek na registraci svému manažerovi.

Důležité jsou pro nás další dva iterátory. První z nich se stará o příjem námi definované zprávy TextNaVelka. Princip a výhody iterátorů v prostředí CCR jsme si popsali v kapitole 3.3.6, teď vidíme jejich praktické užití.

V prvé řadě si všimněme, že je procedura označena atributem Exclusive. Tím dáváme CCR najevo, že budeme chtít tento úsek kódu vykonávat jednotlivě. Zde nás k tomu vede přístup k privátním proměnným. Jediným vstupním parametrem je instance třídy námi definované zprávy, která obsahuje tělo zprávy (obsahem je zaslaný text) a PortSet, na který požadujeme zaslat odpověď. V těle metody je pak pouze sled běžných triviálních operací, včetně jediné „užitečné“, kterou je převod přijatého textu na velká písmena a jeho uložení do vnitřního stavu. Výsledek (kterým je v tomto případě celý stav služby – takové chování jsme od služby požadovali) je zaslán na PortSet získaný z instance zprávy.[31] Nakonec vrátíme vlákno. Při dalším zavolání bude procedura vykonávána znovu od začátku.

Iterátor obsluhující naši druhou zprávu TextNaMala je ještě o poznání jednodušší, mimo inkrementace počítadla odešle zkonvertovaný text na návratový port.

Tím je celá služba dokončena a připravena plnit své úkoly. Abychom mohli demonstrovat její činnost, vytvoříme si další službu (může jít o klidně o standardně vygenerovanou novou službu) nazvanou třeba DemoDSS_ServiceKlient a doplníme pouze řádky z výpisu 16 do souboru DemoDSS_ServiceKlientTypes.cs a řádky z výpisu 17 do souboru DemoDSS_ServiceKlient.cs. Kompletní zdrojový kód takové služby je k dispozici na přiloženém CD.

```
1  using Microsoft.Dss.Core.DsspHttp; //bude umět odpovídat na HTTP GET
2  [DataContract]
3  public class DemoDSS_Service_KlientState
4  {
5      string _text; //budeme uchovávat zkonvertovaný text
6
7      [DataMember]
8      public string Text
9      {
10         get { return _text; }
11         set { _text = value; }
12     }
13 }
14
15 [ServicePort]
16 public class DemoDSS_Service_KlientOperations : PortSet<DsspDefaultLookup, DsspDefaultDrop,
       Get, Subscribe, HttpGet> //musíme přidat HttpGet
17 {
18 }
19 //musíme přidat definici HttpGet
20 public class Get : Get<GetRequestType, PortSet<DemoDSS_Service_KlientState, Fault>>
21 {
22     public Get()
23     {}
24     public Get(GetRequestType body)
```

[31] Pokud bychom chtěli poslat chybovou zprávu, uděláme to rovněž zasláním na stejný port, pouze budeme posílat instanci třídy Fault.

```
25          : base(body)
26      {}
27
28      public Get(GetRequestType body, PortSet<DemoDSS_Service_KlientState, Fault>
            responsePort)
29          : base(body, responsePort)
30      {}
31  }
```

Výpis 16: Podstatné změny v DemoDSS_ServiceKlientTypes.cs

```
1   using Microsoft.Dss.Core.DsspHttp; //bude umět odpovídat na HTTP GET
2   using demo = DemoDSS_Service.Proxy; //pro snadné odkazování na druhou službu
3
4   namespace DemoDSS_Service_Klient
5   {
6       [Contract(Contract. Identifier )]
7       [DisplayName("DemoDSS_Service_Klient")]
8       [Description("DemoDSS_Service_Klient service (no description provided)")]
9       class DemoDSS_Service_KlientService : DsspServiceBase
10      {
11          [ServiceHandler(ServiceHandlerBehavior.Concurrent)]
12          public IEnumerator<ITask> HttpGetHandler(HttpGet httpGet)
13          {
14              demo.TextNaVelka vysledek;
15              yield return _prekladatelPort.TextNaVelka(_state.Text, out vysledek);
16              demo.DemoDSS_ServiceState odpoved = vysledek.ResponsePort;
17              if (odpoved == null)
18              {
19                  LogError("Nepodařilo se provést překlad na velká", (Fault)vysledek.ResponsePort
                        );
20              }
21              _state.Text = odpoved.Text; //pouze pro info-bude se updatovat v browseru
22              demo.TextNaMala vysledek2;
23              yield return _prekladatelPort.TextNaMala(_state.Text, out vysledek2);
24              demo.TextNaMalaResponse odpoved2 = vysledek2.ResponsePort;
25              if (odpoved2 == null)
26              {
27                  LogError("Nepodařilo se provést překlad na malá", (Fault)vysledek2.
                        ResponsePort);
28              }
29              _state.Text += odpoved2.Text; //pouze pro info-bude se updatovat v browseru
30              httpGet.ResponsePort.Post(new HttpResponseType(_state));
31              yield break;
32          }
33
34          [Partner("Prekladatel", Contract = demo.Contract.Identifier, CreationPolicy =
                PartnerCreationPolicy.UseExistingOrCreate)]
35          demo.DemoDSS_ServiceOperations _prekladatelPort = new demo.
                DemoDSS_ServiceOperations();
36
37          /// <summary>
38          /// Service start
39          /// </summary>
```

```
40          protected override void Start()
41          {
42              base.Start();
43              _state.Text="a";
44          }
45      }
46  }
```

Výpis 17: Podstatné změny v DemoDSS_ServiceKlient.cs

Funkce jednotlivých bloků jsme si už popsali, takže jenom připomeňme, že v souboru DemoDSS_ServiceKlientTypes.cs si vytvoříme jednu vlastnost, která bude součástí stavu služby, abychom tak mohli sledovat její změny. Dále zajistíme, že služba bude přijímat zprávy typu HttpGet, musíme tedy definovat typy zasílaných a návratových zpráv včetně konstruktorů a tento typ také zařadit mezi přijímané zprávy hlavního portu.[32]

V souboru DemoDSS_ServiceKlient.cs pak vytvoříme obslužný iterátor pro zprávy typu HttpGet. V něm pouze postupně odešleme požadavek na konverzi textu na velká písmena a po obdržení odpovědi[33] odešleme další požadavek na konverzi na malá písmena.

Dále na řádku 43 nastavíme počáteční text, abychom nevolali konverzi s prázdným řetězcem.

Novinkou je zde na řádcích 33 a 34 vytvoření portu stejného typu jako je hlavní port partnerské služby.[34] Zde si ho výstižně nazveme *překladatel*.

Pokud nyní spustíme „klienta“, běhové prostředí zjistí, že ke své činnosti vyžaduje další službu a pokusí se ji spustit. Pokud pracujeme v prostředí Visual Studia, vidíme v novém okně postupně start obou služeb. Jakmile jsou spuštěny, můžeme v prohlížeči zadat adresu a port klientské služby.[35] Měli bychom vidět seznam služeb běžících v DSS uzlu (ten pro nás rovněž spustí VS). Po kliknutí na název naší služby uvidíme její aktuální stav v podobě XML souboru. Opakovaným obnovováním okna bude docházet k odesílání požadavků na „překladatele“ a tím ke změně vlastnosti Text v XML souboru.

Následně se stejným způsobem můžeme podívat na stav partnerské služby. Zde by se mělo pouze inkrementovat počítadlo a do vlastnosti Text ukládat pouze text velkými písmeny.[36]

Výpis stavů obou služeb v určitý okamžik jejich běhu ukazují obrázky 6 a 7.

Velmi stručně jsme si tedy ukázali, že není nijak složité sestavit plně funkční službu, která je schopna komunikovat s jinými a o jejímž stavu se můžeme přesvědčit webovým prohlížečem. Velmi podobným způsobem bychom také provedli registraci služeb k odběru notifikací. Mohli bychom tak třeba třetí „hlídací“ službou sledovat průběh provádění našich konverzí textu. Některé základní postupy najde zájemce v dokumen-

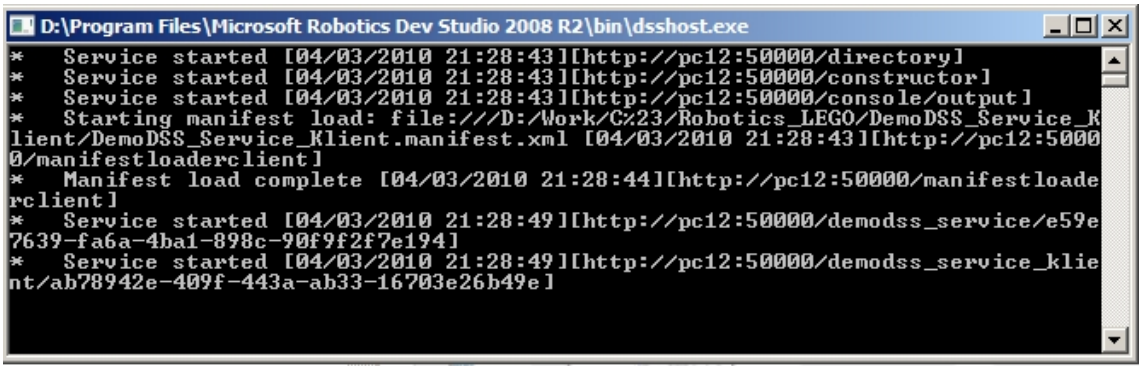
[32] Tento postup včetně konkrétní implementace obslužné metody je převzat z dokumentace k MRDS, konkrétně DSS Service Tutorial 1.

[33] Návrat do iterátoru po obdržení odpovědi je zajištěn příkazem yield **return** ... – viz kapitola 3.3.6

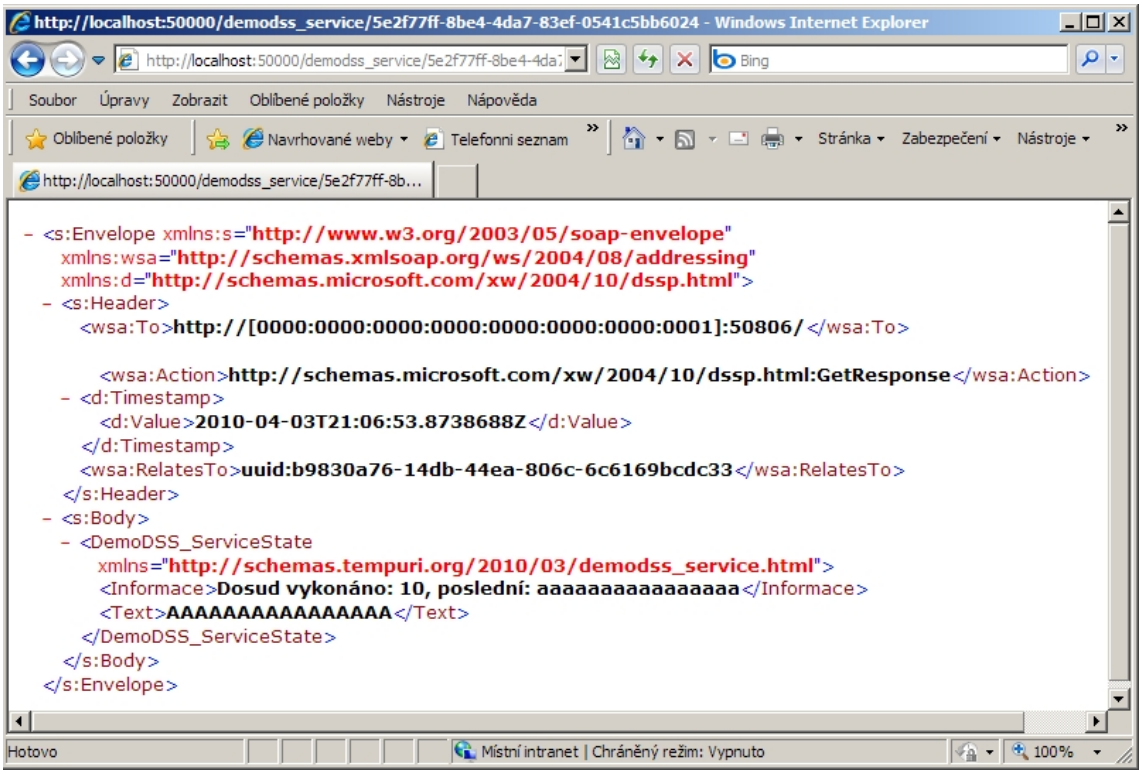
[34] Je zřejmé, že na port partnerské služby nemůžeme odeslat jinou zprávu, než je jí podporovaná.

[35] Adresu předpokládáme na lokálním PC *localhost*, port je defaultně 50000.

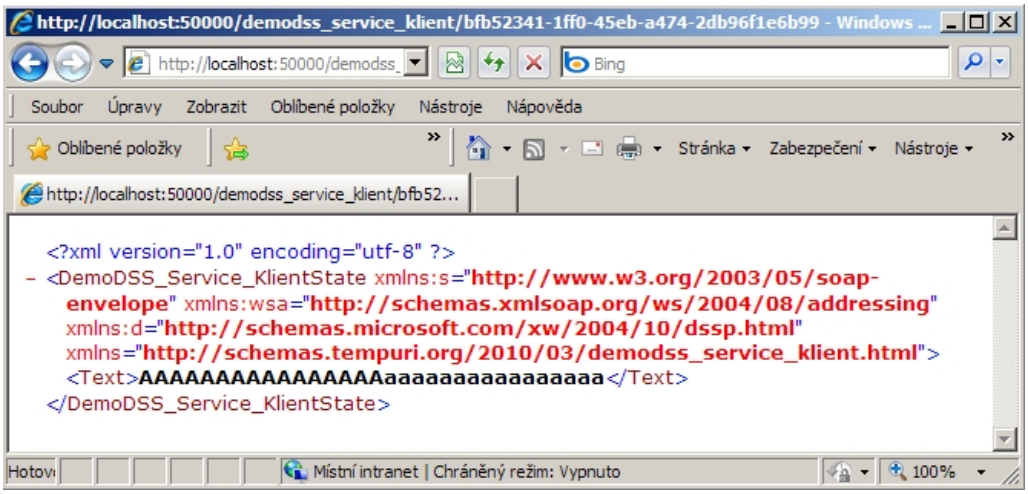
[36] Protože jsme neimplementovali vlastní obsluhu HttpGet, bude použita defaultní, která vrací stav služby v *SOAP* obálce, nicméně pro naše účely je tato forma čitelná a tedy dostačující.



Obrázek 5: Postupné spouštění služeb DSS



Obrázek 6: Běžící klient



Obrázek 7: Běžící překladatel

taci MRDS, ale asi nejlepším vodítkem pro vlastní návrhy je analýza již hotových řešení, kterých lze již dnes najít dostatek na internetu, za všechny třeba [19].

5 Visual Programming Language (VPL)

5.1 Základní informace ke způsobu výkladu

Nyní se dostáváme k samotnému jazyku VPL. Pro čtenáře, který prošel i „nepovinné“ kapitoly CCR a DSS, bude řada rysů tohoto jazyka povědomá a mnohem snadněji pochopí použité algoritmy a programové konstrukce.

Protože autor předpokládá, že ne každý bude mít chuť a trpělivost procházet zdrojové kódy C# a přesto by se chtěl s VPL seznámit, budou příklady podrobně komentovány i s ohledem na jejich chování „pod povrchem“.

Nejprve budou prezentovány příklady poměrně jednoduché, ovšem typické pro nějaký rys jazyka, postupně bude náročnost vzrůstat. Ambicí tohoto textu není vytvořit referenční příručku jazyka, proto je doporučeno průběžně nahlížet do dokumentace MRDS. Na místech, kde to bude vhodné pro lepší pochopení tématu, bude čtenář odkázán přímo na konkrétní kapitolu dokumentace.

Z prostorových důvodů také nelze každý krok tvorby programu dokumentovat obrázkem, budou zařazeny pouze u klíčových kroků celého procesu, a předpokládá se tedy, že čtenář bude při aktivním studiu souběžně pracovat i přímo ve vývojovém prostředí. K dispozici na přiloženém CD má rovněž zdrojové kódy všech zde uváděných příkladů. Bohužel, ačkoli VPL je jazyk vizuální, některé vlastnosti, jak uvidíme dále, nelze na obrázcích zachytit. I proto je velmi důležitá vlastní práce v reálné aplikaci, kde je možné chybějící informace zobrazit nebo vyhledat.

5.2 Základní pojmy

5.2.1 Vývojové prostředí

Na obr. 8 vidíme základní obrazovku vývojového prostředí Visual Programming Language,[37] součásti MRDS.

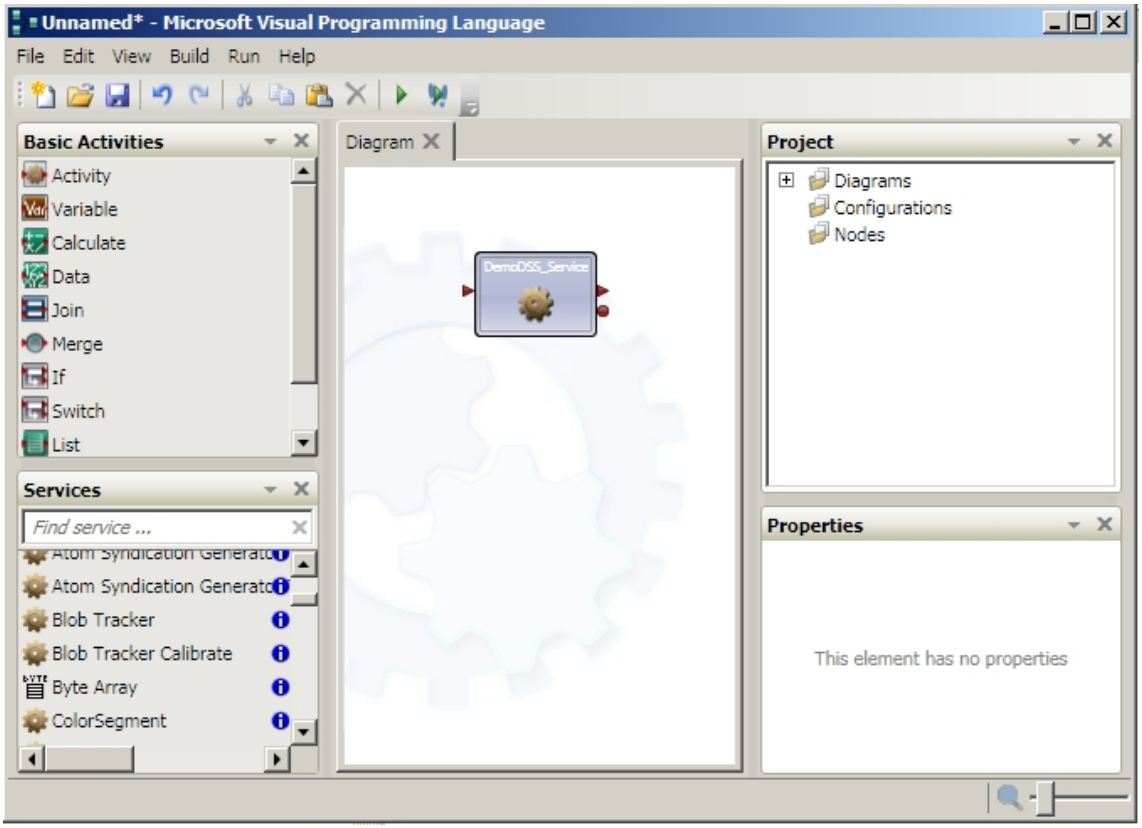
V levé horní části okna máme k dispozici tzv. základní aktivity (basic activities), které tvoří jádro jazyka VPL a které budeme používat během vytváření vlastních aplikací nejčastěji.

V levé dolní části okna pak najdeme tzv. služby (services). Ty představují, jak už název napovídá, služby DSS, které provádějí nějaké užitečné operace a které budeme mít možnost do naší aplikace začlenit.[38] Jde vlastně o DSS služby zapouzdřené v grafickém prvku, se kterým je možné manipulovat prostřednictvím myši a klávesnice. Stejně tak operace známé ze světa DSS, jako je posílání zpráv a odpovědí, registrace a zasílání notifikací apod., jsou ve VPL prezentovány graficky.

Střední část tvoří hlavní pracovní plocha, tzv. Diagram, kam aktivity a služby vkládáme a vytváříme z nich funkční celky. Těchto diagramů můžeme mít pro jednu úlohu vytvořeno i více a lze mezi nimi průběžně přepínat. Pro většinu prvků v diagramu platí,

[37] Protože je vývojové prostředí nazváno stejně jako celý jazyk, budeme dále v textu na místech, kde by hrozilo nedorozumění, používat označení editor VPL, prostředí VPL apod.

[38] Jde o všechny zkompilované DSS služby umístěné v podadresáři *bin* instalace MRDS.



Obrázek 8: Okno vývojového prostředí VPL

že o nich podrobnější informace získáme přidržením kurzoru myši, případně jedním kliknutím. V tom případě se dostupné informace zobrazí vpravo dole a je možné je i editovat.

Poznámka 5.1 Aktivitám a službám, resp. jejich grafickým symbolům v diagramu, budeme dále po vzoru originální dokumentace souhrnně říkat bloky (blocks). Ve většině případů se totiž z hlediska začlenění do aplikace rozdíl mezi aktivitami a službami stírá[39] a můžeme si tak další výklad zjednodušit.

Vpravo nahoře jsou pak vlastnosti projektu jako celku i jeho jednotlivých částí. Zejména zde nalezneme informace o vytvářených diagramech a parametrech pro kompilaci projektu.

5.2.2 Diagram

Už jsme se zmínili o hlavní pracovní ploše VPL, kde vytváříme kód naší budoucí VPL aplikace. Abychom dodrželi konzistenci v pojmech, je třeba říci, že výsledným produktem

[39] Nově definovaná aktivita se po kompilaci stává DSS službou a rovněž editor VPL v některých situacích označuje služby pojmem aktivita.

diagramu bude opět DSS služba, běžně spustitelná v DSS uzlu. Dokonce platí, že každý diagram v jednom konkrétním projektu bude tvořit samostatnou službu.

Poznámka 5.2 V pravém dolním rohu okna editoru najdeme nástroj pro změnu měřítka diagramu (symbol lupy). Ten oceníme zejména u složitějších a tedy „zadrátovanějších" aplikací.

5.2.3 Aktivity

Základním prvkem VPL je tzv. aktivita (activity). Jde o část programového kódu představovanou grafickým symbolem, která nějakým způsobem ovlivňuje činnost programu. Na aktivity se můžeme dívat jako na obdobu instrukcí nebo příkazů běžných programovacích jazyků, přičemž zde program netvoříme sekvenčním řazením těchto příkazů, ale grafickým znázorněním toku dat (zpráv) mezi jednotlivými aktivitami. Mimo větší názornosti ve srovnání s běžným zdrojovým kódem tento přístup také skrývá před uživatelem (programátorem) některé implementační detaily a představuje tak další zefektivnění práce.

Mimo základních aktivit již zabudovaných do VPL máme možnost definovat své vlastní. Opět si je můžeme představit jako analogii podprogramů, procedur či funkcí z jiných jazyků. Také zde tak můžeme aplikaci rozčlenit na dílčí úseky kódu, které můžeme opakovaně volat jako celek, a současně tak kód zpřehlednit, neboť celá aktivita je pak reprezentována jediným grafickým symbolem.[40]

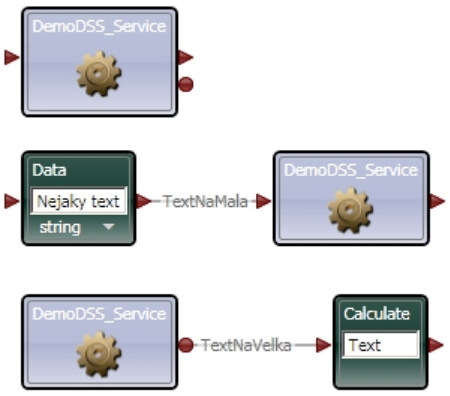
5.2.4 Datový tok

Datovým tokem, či tokem dat (Data flow), budeme chápat posloupnost operací, které se postupně vykonávají na základě požadavků a notifikací předávaných mezi aktivitami a službami. V diagramu tento tok symbolizuje čára spojující jednotlivé bloky. Aby bylo možné snadno zjistit typ požadované operace, mají všechny bloky znázorněny své vstupní, výstupní a notifikační porty (viz DSS *MainPort* služby v kapitole 4.2), a to různými symboly. Šipka (trojúhelníček) směřující dovnitř bloku představuje její vstupní port, šipka směřující ven výstupní port, kolečko pak výstup notifikací.[41] Na obr. 8 jednu takovou službu i s vyznačenými porty vidíme.

Pokud tedy potřebujeme např. odeslat zprávu z jednoho bloku do druhého, stiskneme tlačítko myši nad výstupním portem prvního a táhneme ke vstupnímu portu druhého, kde tlačítko uvolníme. Tím vytvoříme čáru představující onu cestu zprávy. V případě, že dané služby na portu podporují více typů zpráv (opět viz kapitola 4.2), nabídne nám VPL volbu. Typ zprávy se pak zobrazí vedle příslušného „zapojeného" portu.

[40] Od verze R2 lze navíc pro uživatelsky definovanou aktivitu vložit i vlastní ikonu.

[41] Bližší informace o výměně zpráv v kapitole 4, zde pouze stručně. Běžnými porty si jedna služba vyžádá nějakou informaci od druhé a ta následně na výstupní port zašle odpověď. Odpověď je tedy přímo svázána s jedním konkrétním požadavkem. Naopak notifikacemi informuje služba své zaregistrované posluchače o změnách svého vnitřního stavu. Notifikace tedy probíhají automaticky a mimo úvodní registraci je není třeba aktivně vyžadovat.



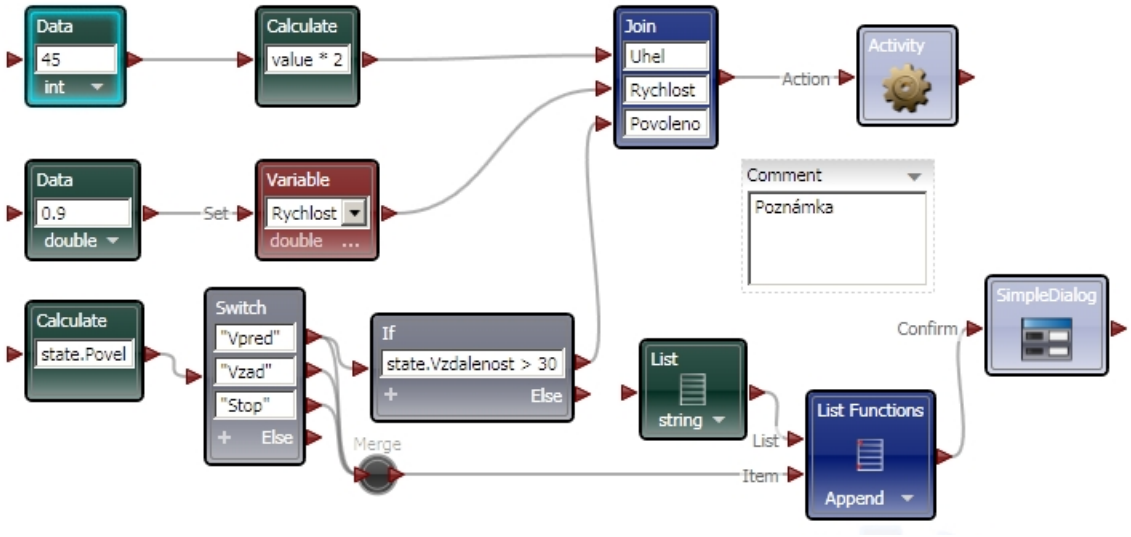
Obrázek 9: Ukázka propojování bloků

5.2.5 Instance aktivit

Zásadním pojmem ve VPL jsou instance aktivit. Vezměme nějakou službu a umístěme ji do diagramu. Opakujme tuto akci znovu a uvidíme, že se nás VPL ptá, zda má vložit původní službu (nabídne se shodný název), nebo přidat novou. Zásadní rozdíl spočívá v tom, že každá instance představuje jednu samostatně spuštěnou službu v rámci DSS uzlu s vlastním vnitřním stavem a vlastním zpracováním zpráv. Podívejme se tedy, proč je možné (a užitečné) vkládat několikrát stejnou instanci téže aktivity.

Pokud si do diagramu umístíme nějakou službu, vidíme, že má vyznačeny všechny své porty (viz obr. 9 nahoře). Nyní si vložme jiný blok (může jít o základní aktivitu ale i jinou službu) a pokusme se vytvořit spojení mezi oběma bloky, jak je znázorněno na obr. 9 uprostřed. Vidíme, že nevyužité notifikační porty už vykresleny nejsou a neměli bychom tedy notifikace jak zaregistrovat a tedy ani dostávat. Jediným řešením zde je právě vložení stejné aktivity ještě jednou. Na nově vložené ikoně již notifikační port k dispozici je. Podobná situace nastane, pokud naopak zapojíme právě notifikační port jako vstup dat pro jiný blok. Pak vidíme (viz obr. 9 dole), že již není vykreslen ani vstupní port. Tím nám editor VPL dává na vědomí, že notifikace nevyžadují žádný požadavek, ale přicházejí spontánně. Tato vlastnost VPL může být na jednu stranu poněkud matoucí, protože stejná aktivita se pak může v diagramu (případně několika diagramech najednou) vyskytovat vícekrát a nemáme tak názornou kontrolu nad tokem zpráv, na druhé straně je však velmi mocným nástrojem ke zvýšení přehlednosti složitějších algoritmů. Asi není těžké si přestavit změť křížících se čar, kterou by byl diagram pokryt, pokud by např. komunikovaly čtyři bloky navzájem každý s každým. Navíc v celém textu práce neustále zmiňujeme asynchronní paralelní provádění operací a toto je nejlepším důkazem, že i VPL je na něm založen a plně ho podporuje. Nic[42] totiž systému nebrání provádět nezávislé části kódu zcela autonomně a „zadrátovaný“ diagram by naopak pro programátora mohl vytvářet iluzi synchronního zpracování.

[42] O některých omezeních se přesto zmíníme v rámci praktických příkladů v kapitole 7.17.



Obrázek 10: Basic Activities

Poznámka 5.3 Všimněme si, že znovu vložená instance služby je v diagramu nazvána stejně. Pokud v některém bloku toto jméno změníme, změní se rovněž ve všech blocích téže instance. Naopak, pokud vložíme instanci novou, editor automaticky změní její jméno tak, aby byl blok odlišitelný. Je proto dobrým zvykem si jednotlivé instance přejmenovat např. v souladu s jejich úlohou v aplikaci.

5.2.6 Proměnné

Běžný program psaný v téměř libovolném jazyce předpokládá existenci nějakých proměnných pro dočasné uchování později zpracovatelných informací. Protože VPL je založen na službách DSS, které nemají žádný společný kontext a komunikují spolu výhradně zprávami, mohou tyto proměnné být pouze součástí stavu služby a jejich hodnoty pak mohou být předávány jako obsah zpráv.

Ve VPL tedy můžeme definovat vlastní proměnné, které se stanou součástí vnitřního stavu budoucí služby, ovšem nemůžeme je sdílet mezi více diagramy. Naopak jejich hodnoty můžeme libovolně předávat prostřednictvím zpráv a také do nich obsah přijatých zpráv ukládat.

Poznámka 5.4 Výše řečené je velmi důležité například v případě, kdy kopírujeme část VPL kódu z jednoho diagramu do druhého. Editor za nás totiž společně s přenesenými grafickými prvky vytvoří i příslušné proměnné, stejně nazvané, ovšem odkazující na zcela samostatné instance. Můžeme tak do kódu zanést velmi obtížně laditelné chyby.

5.3 Základní aktivity

Na obr. 10 vidíme obsah okna *Basic activities*, ovšem už vložený do okna diagramu a vytvářející jakousi část aplikace. Zde poznamenejme, že se nejedná o žádný smysluplný kód, jeho jediným úkolem je demonstrovat způsoby použití těchto aktivit.

K získání detailních informací o každé ze základních aktivit si můžeme v tomto okamžiku vzít na pomoc referenční manuál jazyka VPL, kde najdeme mimo slovního popisu také přesně vyjmenované podporované operace a datové typy. Zde si pouze v krátkosti popíšeme základní použití, případně zmíníme některá omezení.

5.3.1 Data

Jde o blok sloužící k předání konkrétní hodnoty nějakého datového typu. Nejčastěji se používá k inicializaci proměnných, případně předávání konstantních hodnot, známých již při návrhu aplikace.

Na obrázku vidíme, že obsahuje hodnotu 45 typu **int**. Blok obsahuje vstupní i výstupní port, ovšem provádí pouze jedinou operaci, a to předání své hodnoty na základě příchozího požadavku na výstupní port. Typ vstupní operace nemusí být specifikován, pouze se připojí tok dat.

Poznámka 5.5 Zde si všimněme zajímavé situace. Do vstupního portu není zapojena žádná čára, žádný požadavek tedy nepřichází. Jde o další ze specifických rysů VPL. Pokud totiž „zapojíme“ pouze výstupní port, předpokládá se automaticky, že příslušný požadavek má být vyslán okamžitě po startu služby, nevyžaduje-li ovšem nějakou hodnotu. Pohybujeme se v prostředí asynchronních paralelních operací, proto se takto budou chovat všechny aktivity, které stejným způsobem ponecháme bez připojeného vstupního portu, a všechny požadavku se vyšlou současně.[43]

5.3.2 Calculate

Tento blok má za úkol provést nějakou základní operaci s předanými daty a výsledek poslat na svůj výstupní port. Seznam operací a způsob jejich zápisu opět nalezneme v referenční příručce. Nejčastějším použitím aktivity Calculate je úprava hodnot proměnných před nebo po odeslání zprávy, případně výběr a předání vhodné proměnné ze sdruženého toku dat (viz *Join* 5.3.7).

Zde na obrázku se hodnota vstupní proměnné value zdvojnásobí a pošle dál.

5.3.3 Variable

Pokud potřebujeme uložit nějakou hodnotu pro pozdější použití, použijeme blok *Variable*. Z rozbalovacího seznamu si buď vybereme již vytvořenou proměnnou, nebo klepnutím na symbol tří teček vyvoláme dialog s možností založení nové. Pod názvem proměnné vidíme i její datový typ, což přispívá k eliminaci chyb z nepozornosti.

[43] Platí to ovšem pouze pro hlavní diagram, nikoli např. pro uživatelsky definované aktivity, jak uvidíme dále.

Aktivita poskytuje pouze dvě operace Get a Set. Pozor na to, že v obou případech se požadavek posílá na vstupní port a výsledná hodnota na výstupní. Občas totiž působí poněkud matoucím dojmem označení Get mezi dvěma bloky, přičemž skutečná hodnota se předává někomu jinému z výstupního portu.

Na obrázku tedy je hodnota *0.9* typu Double vložena do proměnné Rychlost a tato hodnota je odeslána dál.

Současně si také musíme vysvětlit, že k obsahu proměnné nemusíme přistupovat pouze přes blok *Variable*. Na příkladu druhého bloku *Calculate* vlevo dole vidíme převzetí hodnoty proměnné Povel. To, že se jedná o proměnnou, poznáme podle prefixu state.[44]

Poznámka 5.6 Případ state.Povel rovněž demonstruje způsob, jakým přistupujeme obecně k hlouběji zanořeným proměnným. Ve zprávách lze totiž předávat i složitější proměnné (třídy) obsahující další členské proměnné.

5.3.4 Switch

Jde o obdobu příkazu *switch* z běžných programovacích jazyků a umožňuje větvit datový tok podle obsahu konkrétní proměnné. Větve můžeme libovolně přidávat, přičemž ale vždy platí, že další směr toku se uplatní pouze v případě úplné shody vstupní proměnné a hodnoty uvedené v poli příslušné větve.

Poznámka 5.7 Všimněme si způsobu zápisu textových řetězců. Pro odlišení od názvů proměnných je píšeme v uvozovkách. Toto ovšem platí pouze tam, kde je záměna možná. Například do bloku *Variable* typu String ukládáme text přímo.

Další tok dat pokračuje stejně, jako by *Switch* v cestě zařazen nebyl, tedy zůstává stejný typ zprávy i obsah.

5.3.5 If

Jde o klasické větvení datového toku na základě splnění nějaké podmínky. Na rozdíl od *Switch* se zde neporovnává hodnota v poli se vstupní hodnotou, ale určuje se libovolná logická podmínka. Větve se mohou libovolně přidávat, přičemž platí, že se ve vyhodnocování postupuje odshora dolů. Teprve pokud nevyhoví podmínka žádná, tok pokračuje od portu označeného *else*.

Poznámka 5.8 Vidíme, že zde vlastně o žádné větvení nejde, pouze se další tok dat podmiňuje. Nemusíme tedy za každou cenu zapojovat všechny porty všech bloků, případně uzavírat všechny alternativní cesty. V případě asynchronního řízení je běžné, že probíhá více operací současně a například notifikace, které přicházejí „samy od sebe“ lze takto nejlépe filtrovat. Pokud podmínce nevyhoví tato, třeba jí vyhoví ta příští. Na námi zobrazeném příkladu pochopitelně takto nastavená podmínka přijde ke slovu pouze jednou, pak už se do tohoto bodu program nikdy nedostane, protože nemá odkud přijít zpráva s požadavkem.

[44] Označení je opět konzistentní s koncepcí proměnných jako součástí vnitřního stavu (state) služeb.

5.3.6 Merge

Pokud si vyzkoušíme do již obsazeného vstupního portu nějaké aktivity připojit další požadavek, zjistíme, že čáru nemůžeme na místě vstupního portu cílové aktivity napojit. Občas (spíš často) se ovšem vyskytne potřeba spojit víc požadavků[45], přičemž nám nezáleží na tom, která zpráva přijde dřív, chceme pokračovat ihned po obdržení té první.

5.3.7 Join

Podobnou funkci, tedy sloučit tok, má i tato aktivita, ovšem na rozdíl od *Merge* čeká, až má k dispozici zprávy na všech svých vstupních portech. Teprve potom vyšle zprávu na svůj výstupní port. Jejím obsahem je pak sloučený tok ze všech toků vstupních, přičemž pro odlišení v další fázi zpracování jsou původní zprávy zapouzdřeny a označeny názvy uvedenými v příslušných polích *Join*. V našem případě tedy teprve po dokončení průchodu všech tří dílčích větví na vstup *Join* dojde k odeslání požadavku na aktivitu *Activity*.

Poznámka 5.9 Pro efektivní práci s prostředím VPL nám pomůže vědět, že když pokus s připojením na již obsazený port zopakujeme, ale tentokrát tlačítko myši uvolníme kdekoli nad čarou vedoucí do tohoto portu, nabídne nám systém spojení s již existujícím tokem, dle naší volby buď *Merge* nebo *Join*.

Poznámka 5.10 Pro správnou funkci *Join* je ovšem potřeba, aby vstupní větve nebyly zcela nezávislé, protože pak by mohly být vykonávány v samostatných vláknech a nikdy se nepotkat. Například nelze takto spojovat zprávy od dvou notifikací. Nejčastěji tedy tento způsob sdružení toků použijeme tam, kde nějaký původní požadavek je rozdělen na více různých cest, jejichž doba zpracování se může i výrazně lišit, a my potřebujeme, aby tok pokračoval až po splnění všech dílčích úkolů. Vzpomeňme si třeba na příklad s mailem z kapitoly 3.2.

Poznámka 5.11 V posledních bodech týkajících se směrování toku dat vidíme opět velmi názorně půdorys CCR, konkrétně třídy Receiver a jejich několika variant, které jsme si představili v kapitole 3.3.5.

5.3.8 Remark

Tady asi není třeba příliš vysvětlovat, snad jen, že vhodný komentář jistě pomůže lepšímu pochopení funkce, ovšem „překomentovaný“ kód, zvláště ve vizuálním jazyce, může působit spíše kontraproduktivně.

5.3.9 List

Tento blok představuje obecný seznam nějakých položek, s kterými můžeme manipulovat. Položky nemusí být stejného datového typu a lze je dynamicky vkládat i vybírat.

[45] Velmi podstatnou podmínkou je, že musí jít o zprávy stejného typu, jinak editor spojení nedovolí.

5.3.10 ListFunctions

Prostřednictvím tohoto bloku můžeme provádět manipulace se seznamy různých položek. Trošku atypicky vypadá jeho zapojení do datového toku, ale princip je poměrně jednoduchý. Má dva vstupní porty. Na první z nich (*List*) připojíme nějakou instanci seznamu a na druhý *Item* pak záznam, který hodláme nějak zpracovat. Operaci, kterou požadujeme, vybereme z nabídky ve spodní části ikony. Výsledek provedené operace pak v podobě nového seznamu (*List*) obdržíme na výstupním portu.

5.3.11 Simple Dialog

Přestože nepatří mezi *Basic Activities*, ale najdeme ho mezi službami v levém spodním okně, svým významem do této skupiny patří. Velmi užitečný nám může být při ladění aplikací. Jde o velmi jednoduché okno s textem, který se zobrazí při příchodu zprávy na jeho vstupní port. Typ zprávy také určí vzhled okna (informační, výstražné, potvrzovací). Na výstupní port pak dorazí zpráva v okamžiku, kdy uživatel okno zavře (podle způsoby zavření se generuje i obsah návratové zprávy).

Poznámka 5.12 Takto vyvolané okno má nastavenu životnost pouze 60 sekund. Pokud uživatel do této doby nezareaguje, okno se automaticky zavře a na výstupní port přijde zpráva s hodnotou odpovídající stisknutí tlačítka *Storno*. Toto je třeba vzít v úvahu při psaní částí kódu kritických zejména z hlediska bezpečnosti, například robot dočasně zastavený na základě nějaké výstražné zprávy se může automaticky znovu rozjet.

5.3.12 Activity

Jde o novou instanci vlastní, uživatelsky definované, aktivity. Jde vlastně o novou službu, kterou hodláme do diagramu včlenit. Tvorbou těchto aktivit se budeme zabývat při tvorbě robotických VPL aplikací, například v kapitole 7.17.

5.4 Zdrojový kód

Vlastní projekt VPL aplikace je uložen ve speciálních souborech čitelných pouze v editoru VPL, nicméně v principu jde o komprimovaný XML soubor. Součástí projektu jsou potom případné další XML soubory, tentokrát volně editovatelné, které se do adresáře projektu zkopírují v případě, že projekt využívá některou službu odkazující na tento XML soubor.

Poznámka 5.13 To s sebou přináší výhodu vlastních nezávislých úprav těchto souborů, naopak ale znemožní, aby se do aplikace promítly změny provedené v původních konfiguračních XML použité služby. Musíme s tím tedy při návrzích počítat a mít neustále na paměti, že ačkoli odkazujeme na nějaký společný XML soubor pro celé MRDS, pracujeme vlastně pouze s jeho lokální kopií. Konkrétní příklady si ukážeme v kapitole 7.6.1.

5.5 Spuštění aplikace

Po dokončení návrhu je možné novou službu přímo ve vývojovém prostředí VPL spustit. Ačkoli existuje i možnost kompilace projektu do C# a tím vytvoření běžné DSS služby, je běh VPL programů ve vývojovém prostředí interpretovaný. To mimo jiné znamená, že lze spustit i aplikaci, která obsahuje dílčí chyby. Ty se pak projeví teprve v okamžiku, kdy do tohoto bodu datový tok dorazí.

5.6 Ladění

Spuštěnou aplikaci VPL můžeme velmi jednoduše sledovat v okně webového prohlížeče. Znovu zde oceníme původ VPL v DSS službách, neboť ty, jak víme, musí povinně podporovat požadavky HTTP Get. Proto stačí znát identifikátor služby a ve webovém prohlížeči můžeme sledovat její aktuální vnitřní stav. Jak se přesvědčíme v kapitole 7.7, můžeme v případě sofistikovanějších služeb obdržet výsledek i v lidsky přívětivější formě, než je základní informace v podobě XML souboru.

Přímo běhové prostředí VPL však nabízí i vlastní webové rozhraní, jehož prostřednictvím můžeme krokovat přímo VPL diagram, vkládat body přerušení (breakpoints) nebo v reálném čase sledovat stavy proměnných. Příklad jeho využití najdeme v kapitole 5.10.2.

5.7 Kompilace VPL

Výslednou aplikaci máme také možnost zkompilovat do podoby *dll* knihovny a současně uložit zdrojový kód v jazyku C#. Tato operace je ovšem pouze jednosměrná, tzn., že jednou zkompilovaný kód již nemůžeme načíst zpět ve formě VPL (samozřejmě stále můžeme použít původní VPL diagram, pokud jsme neprovedli žádné změny).

Pokud máme aplikaci složitější, zkompilují se samostatně všechny diagramy a uživatelsky definované aktivity.

Poznámka 5.14 Pokud si kompilaci vyzkoušíme, zjistíme, že název výsledné služba je složen z názvu projektu a názvu příslušného diagramu nebo aktivity. Je tedy vhodné si své projekty označovat podle nějakého logického systému, aby bylo možné se ve vzniklých službách později orientovat.

Kompilaci služeb tak použijeme třeba jako výchozí bod při tvorbě vlastní DSS služby, kdy můžeme jednoduše v grafickém prostředí poskládat základní kostru a později v C# doladit detaily, které ve VPL udělat nemůžeme nebo nechceme.

Stejně tak můžeme vytvářet prototypy budoucích služeb, kdy základní funkcionalita je podobá (tu navrhneme ve VPL) a implementační rozdíly vyřešíme v C#.

5.8 První kroky ve VPL

V následujících úlohách se pokusíme navrhnout a oživit několik velmi jednoduchých, zatím nikoli robotických, VPL aplikací, na kterých si ukážeme elementární postupy, omezení a případně chyby, kterých se můžeme dopustit. K jednotlivým úlohám se vztahuje vždy

jeden obrázek zachycující výsledný diagram. Kompletní zdrojové kódy jsou k dispozici na přiloženém CD, název projektu je vždy uveden v popisu obrázku.

5.9 Úloha 1 – Využití dialogů

5.9.1 Zadání

Umožnit uživateli spustit součet dvou čísel a informovat ho o výsledku.

Poznámka 5.15 Toto je dobrá praxe i s ohledem na programování robotických aplikací. Paralelní běh služeb totiž umožňuje zaslat zprávy některým službám ještě dříve, než se jiné spustí, případně zinicializují. To v případě robota může znamenat nepředvídatelné výsledky, proto je lepší datový tok spustit ručně až po kompletní inicializaci celé aplikace.

5.9.2 Postup řešení

- **Zobrazení výzvy**
  První část úkolu, start datového toku, vyžaduje, abychom nějakým způsobem zobrazili výzvu ke spuštění aplikace a následně zpracovali reakci uživatele. V přehledu jazyka VPL v kapitole 5.3 jsme se mimo jiné seznámili se základním prvkem *Simple Dialog*, který je schopen zobrazit požadovaný text a vrátit nějakou odpověď.

  Jako první tedy do diagramu umístíme novou instanci tohoto prvku.

  Nyní potřebujeme, aby zobrazil nějaký vhodný text. Víme, že text je znám již během návrhu a dále ho nehodláme měnit, není tedy zapotřebí zakládat novou proměnnou, ale můžeme text uložit do prvku *Data* typu String. Vložíme tedy do diagramu tento prvek, zvolíme typ a vyplníme text.

  Nyní tyto dva prvky propojíme, z nabídnutých možností typu požadavků vybereme *ConfirmDialog* a jako hodnotu zvolíme možnost *value*, která představuje hodnotu předávaného řetězce (zpráva jinou informaci ani neobsahuje).

  Poznámka 5.16 Bohužel poměrně nepraktickou vlastností editoru VPL je, že jako výchozí typ odpovědi nabízí *Fault* a nikoli *Success*, ačkoli je tato volba určitě častější. Je tedy zapotřebí jedno kliknutí navíc, ale zejména je třeba vždy volbu zkontrolovat.

  V této fázi už můžeme zkusit aplikaci spustit. Podle očekávání se zobrazí potvrzovací dialog.

- **Inicializace vstupních hodnot**
  Zatím nepotřebujeme provádět nějaké komplikované úlohy s pamětí, takže nám pro vložení čísel opět postačí další dva prvky *Data*, tentokrát ale nějakého číselného typu, třeba **double**.

- **Větvení podle volby uživatele**

  Požadujeme ovšem, aby se součet spustil až v okamžiku, kdy operaci uživatel potvrdí, proto nemůžeme ponechat vstupní porty volné (systém by spustil jejich větve datového toku současně se zobrazením dialogu), ale musíme je navázat na výsledek dialogu. Pokud by nám stačilo prosté odklepnutí libovolného tlačítka,

můžeme prostě odeslat zprávu o úspěšném zavření dialogu dál. My ovšem chceme testovat, jaké tlačítko uživatel stisknul.

Použijeme tedy prvek *If*, ve kterém otestujeme hodnotu zaslanou v návratové zprávě. Když klepneme do políčka pro podmínku, nabídne nám editor na výběr z aktuálně dostupných testovatelných hodnot a proměnných. Nás zajímá proměnná *Confirmed*. Pokud uživatel stisknul *Ano*, bude mít hodnotu *true*, jinak false.

Poznámka 5.17 Ačkoli poskytuje dialog zprávy typu *Success* a *Fault*, neznamená to, že odpovídají volbě uživatele. Jde pouze o pozitivní nebo negativní potvrzení provedené operace.

- **Výpočet**

  Výstup z podmínky nyní musíme směřovat na vstup obou prvků *Data*. Můžeme to provést přímo propojením dvěma čarami, neboť na rozdíl od vstupních portů, z výstupních portů můžeme zasílat více zpráv různým partnerům.

  V další fázi potřebujeme provést součet obou hodnot. Samozřejmě sáhneme po *Calculate*, ovšem musíme na jeho vstup přivést obě hodnoty současně. Nelze tedy použít *Merge*, který pošle pouze jednu (tu rychlejší) zprávu, ale *Join*.

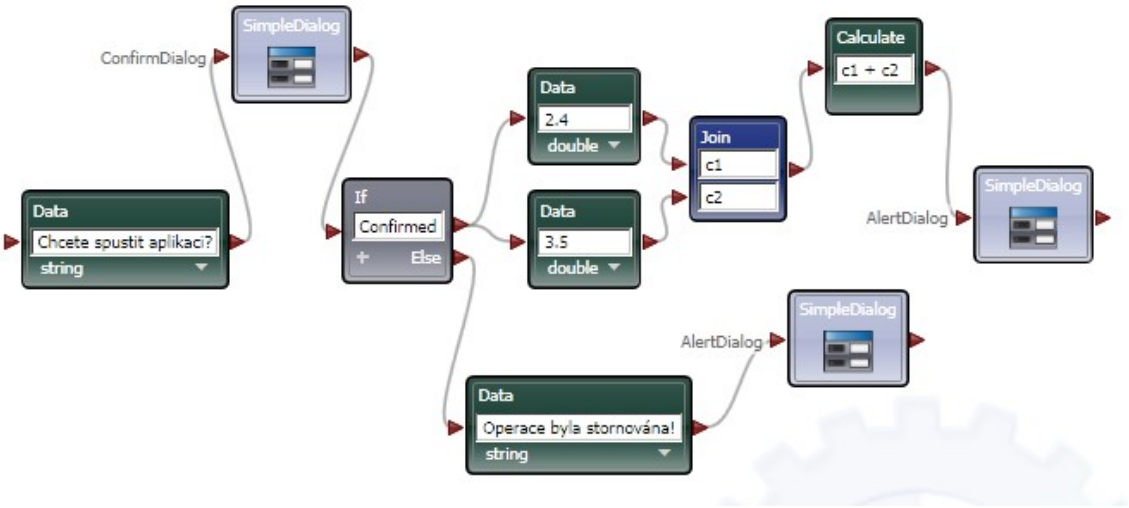
  Poznámka 5.18 Zde můžeme využít jedné z předností[46] editoru. Když totiž přetáhneme myší nějaký blok na čáru (datový tok), bude do tohoto toku automaticky vřazen. V našem případě tedy ušetříme jedno ruční kreslení čáry. Obě větve si pojmenujeme podle hodnot, které předávají (není to nutné, ale je to dobrý návyk pro složitější diagramy) a výstup *Join* spojíme se vstupem *Calculate*. Zde už víme, že stačí kliknout do políčka a editor nám nabídne mimo jiné i právě definované názvy pro dvě posílané proměnné. Stačí tedy do políčka zapsat jejich součet.

- **Zobrazení výsledku**

  Na závěr nám ještě zbývá informovat uživatele o výsledku. Opět se nabízí *Simple Dialog*. Tentokrát budeme text sestavovat dynamicky, nemůžeme tedy použít konstantní hodnotu z prvku *Data*. Pokud výsledek součtu přivedeme na vstup dialogu, zjistíme, že jedinou rozumnou možností je zvolit *value*, což bude pravděpodobně onen výsledek součtu. Je zde ovšem ještě nenápadné zaškrtávací políčko *Edit values directly* v levém dolním rohu (případně při následné editaci spojení v pravém spodním okně editoru). Pokud toto políčko zaškrtneme, můžeme následně jako hodnotu zobrazeného textu zadat libovolný výraz. Protože je možné zapisovat i proměnné, hrozí možnost záměny textového řetězce a ten tedy v tomto případě musíme psát v uvozovkách. Zapíšeme tedy výraz *"Výsledek je: "+value*.

- **Dokončení**
  Naše první aplikace je tak dokončena a můžeme ji spustit.

[46] V případě velmi zaplněného okna diagramu se to ovšem může jevit i jako nevýhoda, protože hrozí i neúmyslná změna toku dat při neopatrném přesunu bloků nad již existující čáru.



Obrázek 11: VPL_SimpleDialog – výsledný diagram

Pro pořádek by bylo vhodné nějak potvrdit i negativní volbu uživatele. K tomu účelu můžeme na větev nevyhovující podmínce navázat další dialog, tentokrát typu *AlertDialog*, opět přes *Data* nastavující vhodné upozornění.

5.10 Úloha 2 – Použití časovače ke vložení zpoždění

5.10.1 Zadání

Rozšířit předchozí aplikaci o opakované přičtení hodnoty k předchozímu výsledku, ovšem s prodlevou tří sekund. Po každém výpočtu se uživateli zobrazí zpráva s výzvou k pokračování nebo ukončení výpočtu.

5.10.2 Postup řešení

- **Využití částí předchozího úkolu**
  Tento úkol samozřejmě vychází z předchozího, takže budeme jistě moci využít i některé části již vytvořeného kódu. Určitě to bude vstupní část s podmínkou a spuštění výpočtu, stejně jako závěrečná část s vydáním výsledku nebo ukončením výpočtu.

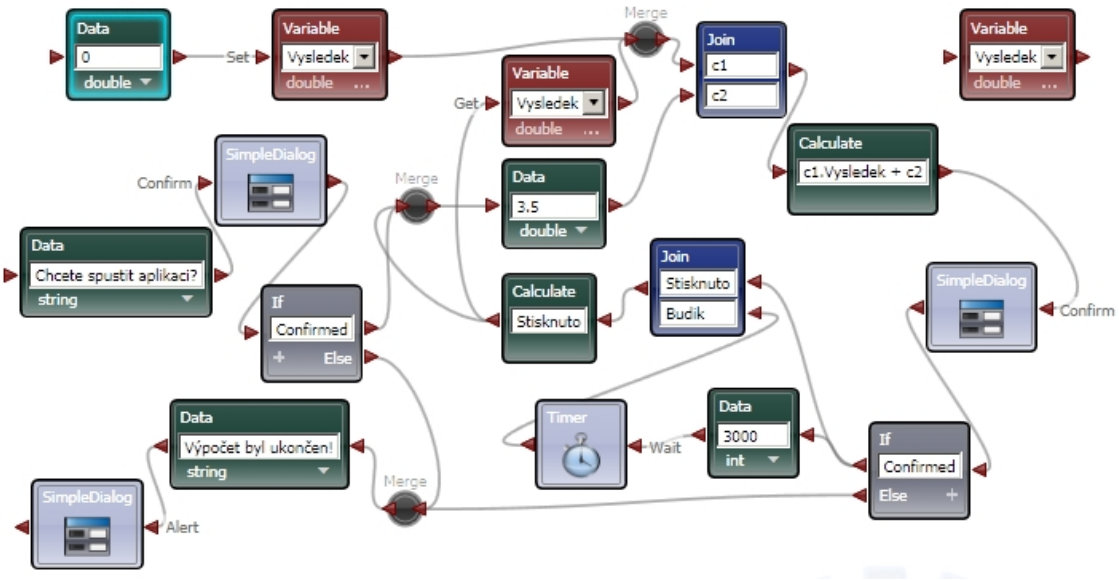
  Rozdíly najdeme dva. Za prvé si musíme pamatovat výsledek předchozího výpočtu, proto musíme použít nějakou proměnnou. Vložíme tedy do diagramu blok *Variable* a poklepáním na tři tečky v jeho ikoně přidáme novou proměnnou typu **double**. Za druhé budeme potřebovat nějak zařídit prodlevu ve vykonávání kódu. Naštěstí máme k dispozici připravenou službu *Timer*, kterou najdeme v levém okně mezi službami, rovněž tu tedy vložíme do diagramu.

- **Inicializace hodnot**
  Nejprve musíme nastavit počáteční hodnotu výsledku, což bude nula. Proto vložíme další blok *Data*, nastavíme ho na stejný typ jako naši proměnnou a vložíme hodnotu 0. Následně tento blok propojíme s *Variable* a protože chceme hodnotu nastavit, zvolíme typ zprávy *Set*.

- **Výpočet**
  Jelikož jedním ze sčítanců bude samotný výsledek, můžeme odstranit jeden z bloků *Data* a místo něj připojíme na vstup *Join* právě výstup z naší nové proměnné. V tom okamžiku ovšem editor upozorní vykřičníkem na to, že hodnoty *c1* a *c2* v bloku *Calculate* jsou rozdílného typu. Při přidržení myši nad vykřičníkem si ověříme, že zatímco *c2* je typu **double**,*c1* je typu Vysledek. Je tomu tak proto, že zpráva od bloku *Data* obsahuje pouze vlastní hodnotu, zatímco blok *Variable* odesílá odpověď na požadavek *Set*, v němž je proměnná zapouzdřena. Blok *Join* pak celou tuto zprávu označí jako *c1* a my si z ní musíme hodnotu proměnné „vytáhnout“ zápisem *c1.Vysledek*.

  V této fázi už můžeme zkusit aplikaci spustit. Uvidíme, že proběhne jeden výpočet s vydáním výsledku.

- **Zařazení zpoždění**
  Abychom mohli výpočet opakovat, musíme dát uživateli možnost další postup ovlivnit. Proto změníme typ výsledného dialogu na *ConfirmDialog* a vrácenou hodnotu otestujeme stejně jako při spuštění aplikace.



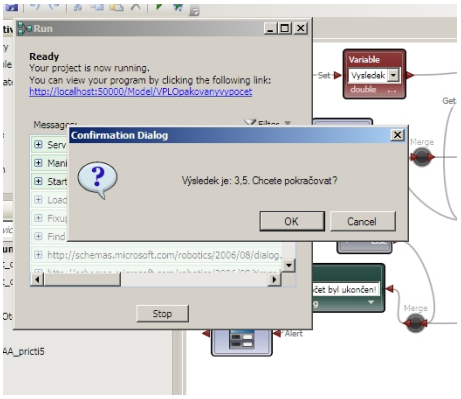
Obrázek 12: VPL_Opakovany_vypocet – výsledný diagram

Poznámka 5.19 Nabízelo by se jednodušší řešení, totiž směrovat výsledek opět do vstupního dialogu a dál opakovat výpočet, nicméně tím bychom přišli o možnost zařadit zpoždění do fáze mezi zobrazením výsledku a zahájením dalšího výpočtu, případně bychom museli testovat, zda se jedná o první nebo opakovaný průchod, což by diagram zbytečně komplikovalo.

Po splnění podmínky v bloky *If* bychom tedy měli zařadit požadované zpoždění. *Timer* nabízí nejjednodušší použití tím, že na jeho vstup přivedeme zprávu *Wait* s délkou prodlevy v milisekundách a jeho výstup odešle zprávu Success teprve po uplynutí nastaveného zpoždění. Zařadíme tedy další blok *Data*, tentokrát typu **int**, nastavíme hodnotu zpoždění na 3000 ms a tento blok propojíme se vstupem *Timer*.

Pokud se nyní pokusíme výstup z timeru připojit přes *Merge* na vstup bloku *Data* s prvním sčítancem, kam bychom ho logicky zapojit chtěli, zjistíme, že toto spojení není z důvodu různých typů zpráv možné. Máme tedy dvě možnosti. Buď si vytvoříme další blok *Data*, který bude mít stejnou hodnotu jako první sčítanec a spojíme teprve jejich výstupy, nebo se pokusíme získat stejný typ zprávy, který po nás editor v tomto místě chce. Vyzkoušíme si ten druhý způsob, neboť představuje „čistější“ řešení bez zbytečných duplicit v kódu.

Vidíme, že zpráva, ke které se chceme připojit, pochází od vstupního dialogu a prochází nezměněna přes blok *If*. Můžeme se tedy pokusit do tohoto místa zapojit výstup našeho dialogu s výsledkem, opět až za blokem *If*. Zjistíme, že toto spojení už nám editor povolí. Když se ale zamyslíme nad výsledkem, stále postrádáme vložené zpoždění, protože výstup timeru stále nemáme zapojený.



Obrázek 13: VPL_Opakovany_vypocet – zahájení ladění

Ideálním řešením tedy bude opět spojení dvou typů zpráv přes *Join*, čímž zajistíme, že zpráva od dialogu projde teprve ve chvíli, kdy vyprší zpoždění v timeru. Z výstupu bloku *Join* pak už jen odfiltrujeme potřebný typ zprávy a ten zapojíme na původně zamýšlené místo.

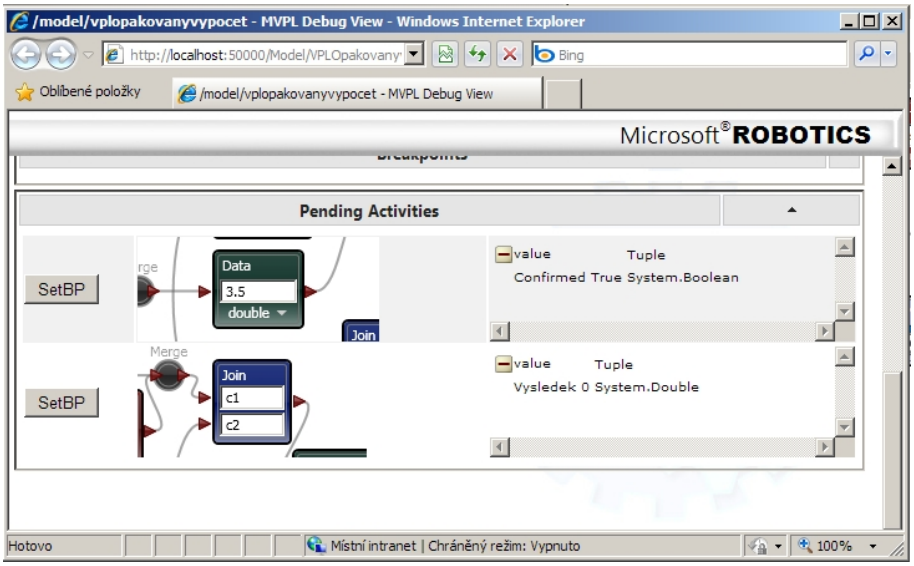
Nyní můžeme opět aplikaci spustit. Pravděpodobně zjistíme, že první výpočet proběhl bezvadně, dokonce máme možnost potvrdit pokračování, ale další výsledek už neuvidíme.

- **Ladění**
  V této fázi, pokud nám není zřejmé, kde je chyba, můžeme přistoupit k ladění aplikace. S výhodou můžeme použít vestavěné webové rozhraní, ve kterém si zobrazíme aktuální stav naší aplikace a můžeme ji dokonce krokovat.

  Na obr. 13 vidíme stav, ve kterém ladění spustíme klepnutím na zvýrazněný odkaz. Otevře se okno našeho výchozího prohlížeče[47], kde vidíme mimo jiné i náš diagram. Detailní popis všech možností debuggeru je nad rámec tohoto textu, ale v integrované nápovědě MRDS je velmi dobře popsán.

  Pro nás bude v tuto chvíli nejvhodnější běh programu přerušit a dál krokovat po jednotlivých zprávách mezi aktivitami. Ve spodní části okna prohlížeče se nám budou postupně zobrazovat výřezy diagramu i s aktuálními hodnotami předávaných parametrů ve zprávách (viz obr. 14). Postupným krokováním dojdeme až do bloku *Join*, který spojuje předchozí výsledek a druhého sčítance. Zjistíme, že z tohoto místa aplikace dále nepokračuje. Pokud jsme si dobře všímali jednotlivých kroků, určitě jsme zjistili, že do bloku *Join* jsme přišli pouze jednou cestou, tou od timeru. Druhá větev tedy stále čeká na příchozí zprávu, aby měl *Join* obě zprávy k dispozici a mohl pokračovat dále. Chybu tedy musíme hledat tady.

[47] Je doporučeno použití Internet Exploreru firmy Microsoft, u alternativních není garantována plná funkčnost.



Obrázek 14: VPL_Opakovany_vypocet – stav během krokování

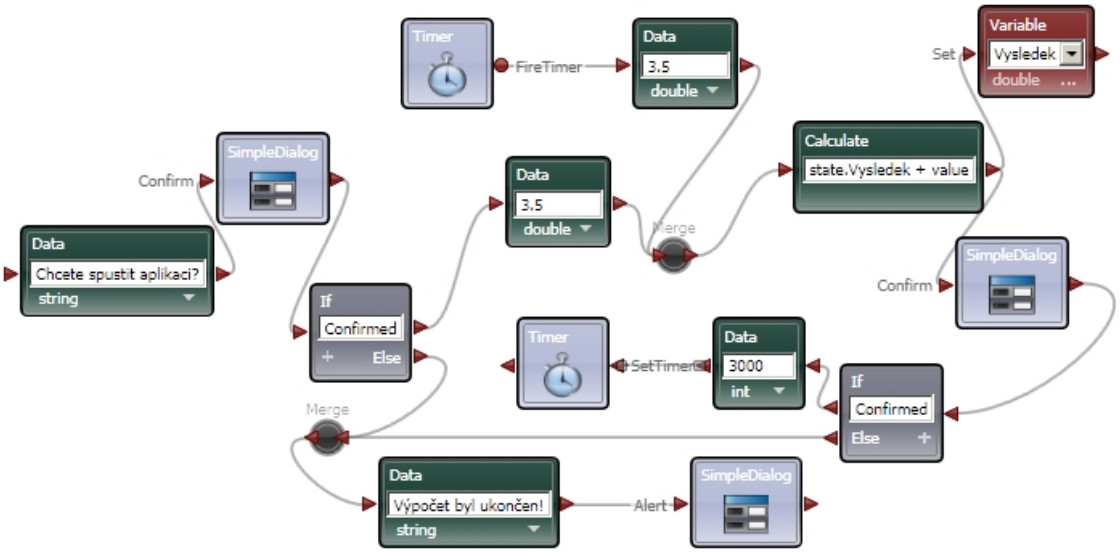
Znovu se podívejme na to, jaká zpráva do této větve vstupuje. Jde o hodnotu proměnné *Vysledek*, ovšem vyvolanou požadavkem *SetValue* při inicializaci aplikace od bloku *Data*. Je tedy zřejmé, že tato zpráva přijde pouze jednou, při startu aplikace a dále už nikdy. Musíme tedy nějak zajistit, aby se zopakovala při každém průběhu výpočtu. Řešení je velmi jednoduché. Vytvoříme si další kopii bloku *Variable* se stejnou proměnnou a na její vstup přivedeme zprávu z bloku *Calculate*, ale tentokrát vybereme typ požadavku *GetValue*. Výstup bloku pak přes *Merge* přivedeme na požadovaný vstup *c1*.

Poznámka 5.20 Zde platí stejné omezení jako u *Simple Dialogu*. Pokud do 60 s neprovedeme další operaci, bude vygenerována chyba (*Fault*) a běh bude pokračovat jakoby poslední služba/aktivita sama tuto zprávu vyslala. To pak může změnit další tok dat v diagramu.

Nyní můžeme aplikaci znovu spustit, ale zjistíme, že ani tentokrát není plně funkční. Zpoždění sice funguje, ale výpočet vrací stále stejnou hodnotu výsledku.

- **Dokončení**
  V tomto případě asi nebude třeba ladění, abychom chybu našli. Nikde v diagramu jsme totiž neukládali výsledek součtu do příslušné proměnné a ta je tedy stále nulová. Chybu napravíme tím, že vytvoříme další kopii bloku *Variable* s příslušnou proměnnou a na její vstup přivedeme výsledek bloku *Calculate*, který provádí výpočet. Tentokrát zvolíme opět typ *SetValue* a jako hodnotu budeme ukládat *value*, což je výsledek prováděné operace.

  Teprve nyní by měla aplikace fungovat správně. Výsledný diagram zachycuje obr. 12.



Obrázek 15: VPL_Opakovany_vypocet2 – výsledný diagram

Poznámka 5.21 Na obr. 12 taky vidíme, že některé bloky mají zrcadlově obrácené umístění svých portů. Tuto možnost nabízí editor pro zlepšení čitelnosti kódu a můžeme ji aktivovat volbou *Flip Connections* z kontextové nabídky nad daným blokem.

- **Náměty na vylepšení**
  Stejně jako každá jen trochu složitější úloha, ani tato nemá jen jedno řešení. Na obr. 15 vidíme jinou variantu, která řeší stejnou úlohu o něco úsporněji z hlediska použitých bloků. V krátkosti si popíšeme, v čem změny spočívají.

  – Inicializace proměnné
    V tomto řešení nikde nevidíme počáteční inicializaci proměnné na nulovou hodnotu. Proměnné číselných typů totiž nabývají nulových hodnot implicitně, stejně jako booleovské proměnné hodnoty false a textové obsahují prázdný řetězec. Použití tohoto přístupu je na uvážení programátora, přispěje sice k redukci kódu, ale může být i zdrojem chyb. My budeme v následujícím textu všechny proměnné inicializovat explicitně.
  – Přístup k proměnné
    V bloku *Calculate* je použit jiný způsob přístupu k proměnné. Není přivedena blokem *Variable* na jeho vstup, ale odkazována přímo přes stav služby (*state.Vysledek*). Toto řešení budeme dále v textu používat častěji, proto stojí za zmínku.
  – Jiné použití časovače
    Timer umožňuje i jiné použití pro nastavení zpoždění. Po zaslání požadavku typu *SetTimer* s hodnotou zpoždění nemusíme čekat na odpověď na výstupu,

ale můžeme použít další kopii téže instance timeru a čekat na notifikaci. Tu timer vyšle v okamžiku vypršení stanoveného času.

Poznámka 5.22 Dle dokumentace MRDS je doporučeno spíše použití této metody, pro intervaly delší než 60 sekund dokonce ani první variantu použít nelze, systém ohlásí chybu. Na druhé straně je ale třeba si uvědomit, že notifikace budou přicházet při každém vypršení timeru, at'už je nastaven kdekoli v kódu. U složitějších aplikací tedy musíme použít buď více instancí timeru, nebo vložit dodatečnou podmínku do cesty takto generovaného datového toku.[48]

[48]Dalším omezením je to, že notifikace nelze použít v uživatelsky definovaných aktivitách. Toto by se mohlo dle vyjádření autorů (viz [20]) v příštích verzích změnit.

5.11 Úloha 3 – Vytvoření vlastní aktivity – cyklus While

V kapitole 5.2.3 jsme přirovnávali aktivity jazyka VPL k základním instrukcím jiných programovacích jazyků. Při podrobnějším popisu jednotlivých bloků, které máme k dispozici, jsme ale nenašli žádný, kterým bychom mohli provádět nějaké operace opakovaně, tedy v cyklu. Jazyk VPL ovšem poskytuje i tuto možnost, neboť cyklus můžeme provádět přímo vytvořením smyčky v datovém toku s tím, že podmínku opakování vyhodnotíme v bloku *If*. Příklad smyčky použité pro jednoduché počítadlo najdeme např. v [21], definici vlastní aktivity provádějící cyklus *For* pak v [1], kde jsou v této souvislosti rovněž diskutována pravidla pro správné vytváření cyklů. Nesmíme totiž zapomenout na to, že se stále pohybujeme v prostředí paralelního zpracování zpráv. Použití takového bloku ilustruje obrázek 16.

My si v následující úloze vyzkoušíme vytvoření obdobné aktivity, tentokrát pro cyklus *While*.

5.11.1 Zadání

Vytvořit vlastní aktivitu, která bude řídit vykonávání kódu na principu cyklu *While*.

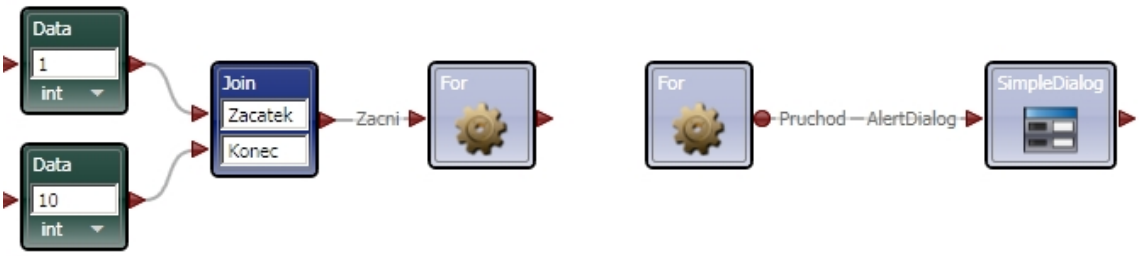
5.11.2 Postup řešení

- **Analýza problému**
  Nejprve se zamysleme, jak cyklus *While* funguje. Na začátku bloku cyklu stanovíme podmínku, při jejímž splnění se tělo bloku jedenkrát provede a následně se podmínka opět vyhodnotí. Po prvním negativním vyhodnocení podmínky je vykonávání cyklu ukončeno.

  Když srovnáme princip cyklu *While* s *For*, je na první pohled zřejmý zásadní rozdíl. Zatímco u cyklu *For* známe dopředu přesný počet opakování a můžeme ho tedy vyhodnocovat přímo v těle aktivity, neboť nijak nesouvisí s předmětem činnosti kódu smyčky, u *While* je tomu jinak. Zde ověřujeme, zda kód cyklu provedl takovou změnu, která nějak ovlivní podmínku pro opakování, ta je tedy na předmětu činnosti kódu smyčky (nebo jiné události mimo aktivitu – např. časovač) přímo závislá.

  Tím ovšem musíme už dopředu rezignovat na úplné oddělení logiky cyklu (podmínky) od zbytku kódu a tedy vytvoření univerzálního bloku *While* pro jakýkoli kód cyklu. Máme pouze dvě možnosti. Buď vytvoříme speciální blok *While* pro každou typickou podmínku cyklu (např. velikost nějaké proměnné neklesne pod určitou hodnotu, nějaké dvě proměnné se rovnají apod.), nebo „vyneseme“ podmínku cyklu z těla bloku ven a necháme na tvůrci aplikace, aby ji stanovil sám.

  První řešení se velmi podobá výše zmíněnému bloku *For*, jehož použití v diagramu ilustruje obrázek 16. Nejprve nastavíme vstupní hodnoty proměnných a cyklus spustíme a posléze při každém průchodu cyklem hodnoty aktualizujeme. Vnitřní podmínka bloku *While* pak pomocí notifikací určuje, zda se má cyklus vykonat znovu.



Obrázek 16: Cyklus *For* – princip použití

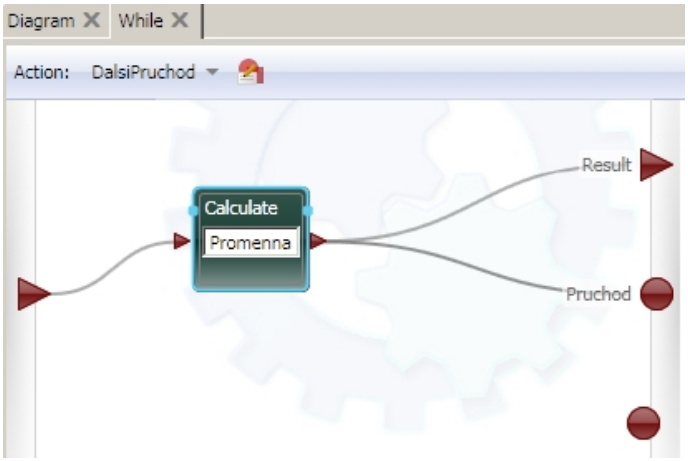
Druhé řešení bude poněkud odlišné. Protože podmínka nebude součástí cyklu, potřebujeme, aby se hodnota příslušných proměnných objevila na notifikačním výstupu naší aktivity *While*. Současně musíme umět cyklus odstartovat a vykonávat další kroky.

- **Vytvoření vlastní aktivity**
  Vlastní uživatelsky definovanou aktivitu vytvoříme velmi jednoduše vložením bloku *Activity* na plochu diagramu. Následně změníme jméno tak, aby vystihovalo účel aktivity. Pokud nyní poklepeme na ikonu vložené aktivity, zobrazí se nám nové okno, podobné jako okno hlavního diagramu. Rozdíl je na bočních okrajích okna, kde jsou znázorněny jednotlivé porty aktivity. Jsme vlastně uvnitř bloku a tyto porty přesně odpovídají těm, na které pak budeme z vnějšku propojovat ve výsledném diagramu.

  Další změnou je menu a tlačítko *Action* u horního okraje okna. Po kliknutí se nám zobrazí jednoduchý dialog, ve kterém můžeme definovat naše vlastní akce a notifikace. Pro každou pak můžeme vytvořit libovolný počet vstupních a výstupních proměnných. Když si vyzkoušíme vytvoření jedné nové akce, zjistíme, že se současně vytváří i nová prázdná plocha v editoru. Každá akce se totiž opět definuje vlastním dílčím diagramem datového toku a je zcela nezávislá na ostatních.

- **Vytvoření akcí**
  Už v analýze problému jsme si definovali akce, které má naše aktivita provádět. Nejprve to bude odstartování prvního průchodu cyklem a následně provedení dalšího průchodu. Vytvoříme si tedy dvě akce a nazveme je *Zacni* a *DalsiPruchod*. Protože současně musíme hlavnímu dialogu zpřístupnit hodnotu proměnné, aby mohla být vyhodnocena podmínka, přidáme do obou akcí i jednu vstupní proměnnou, nazvanou *Promenna*.

- **Diagramy akcí**
  Vlastní činnost aktivity bude přímo triviální. Proměnnou přivedenou na vstup pouze přivede na svůj výstup, čímž dá pokyn k analýze podmínky. Kdybychom ovšem použili běžný výstupní port, asi bychom takovou aktivitu ani použít vůbec nemuseli, její zařazení do diagramu by bylo naprosto zbytečné. Když ale použijeme notifikační výstup, dosáhneme zcela odlišného efektu.



Obrázek 17: VPL_While – akce *DalsiPruchod*

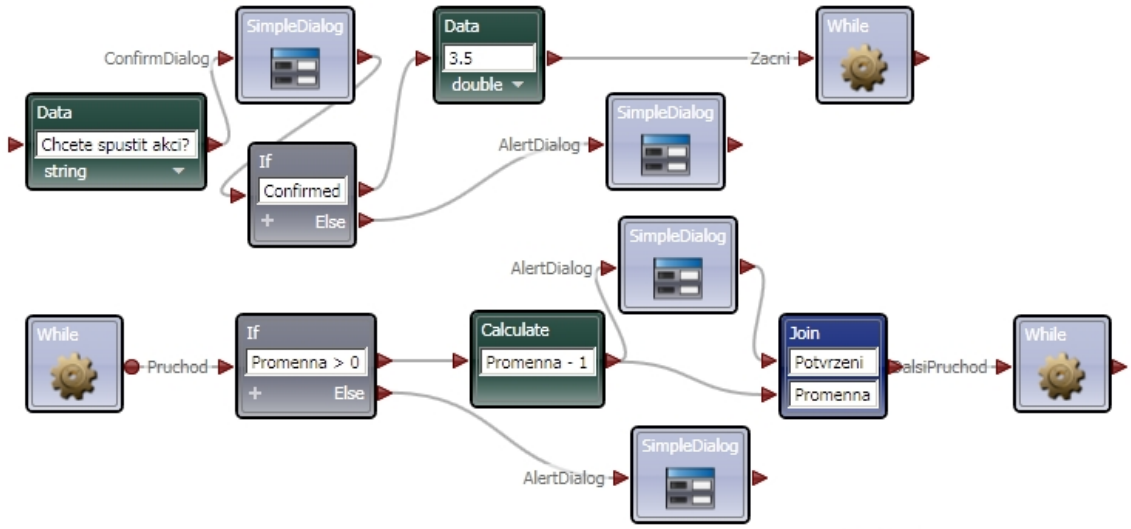
Vložíme tedy do diagramu akce *Zacni* blok *Calculate*, do kterého přivedeme vstupní proměnou, kterou také v poli vybereme, a jeho výstup přivedeme na notifikační port aktivity. Editor se nás zeptá, zda má vytvořit novou notifikaci, což potvrdíme. V dialogu akcí pak pouze přidáme i pro tuto notifikaci novou proměnnou (logicky si ji opět pojmenujeme *Promenna*). Na notifikačním portu ještě nastavíme správně předávání hodnoty bloku *Calculate* do notifikační proměnné.

Naprosto stejně budeme postupovat i u akce *DalsiPruchod*, ale využijeme už existující notifikaci, pro hlavní diagram není podstatné, co vyvolalo pokyn pro další průchod smyčkou. Diagram akcí vidíme na obr. 17.

- **Zařazení do diagramu**
  Nyní můžeme naši aktivitu použít přímo v hlavním diagramu. Vytvoříme si malý demonstrační program, který bude mít za úkol v cyklu odečítat jedničku od hodnoty předané proměnné a při záporné hodnotě cyklus ukončí. Průběžně bude o výsledku informovat uživatele. Vlastní algoritmus si zde nebudeme popisovat, je velmi jednoduchý, pouze se zaměříme na použití nové aktivity.

  Výpočet musíme nějak odstartovat, což provedeme přivedením počáteční hodnoty proměnné na vstup bloku *While* se zvolenou akcí *Zacni*. Nyní musíme do diagramu vložit další kopii téže instance aktivity a tentokrát použijeme její notifikační port jako vstup do vlastního cyklu. Nejprve vyhodnotíme podmínku, následně provedeme vlastní výpočet a zobrazení výsledku a po odsouhlasení uživatelem přivedeme novou hodnotu proměnné opět na vstup *While*, tentokrát ale akce *DalsiPruchod*. Program si můžeme spustit a uvidíme, že opravdu funguje jako cyklus *While*. Výsledný diagram je na obr. 18.

  Poznámka 5.23 Použití dvou samostatných akcí , které dělají v podstatě totéž, nám může připadat zbytečné, ale rozhodně přispěje k čitelnosti kódu, když při opakování cyklu použijeme akci *DalsiPruchod* a ne *Zacni*, ačkoli jejich činnost je totožná.



Obrázek 18: VPL_While – výsledný diagram

- **Zhodnocení**
  Nyní je na místě otázka, zda má vůbec smysl takovouto triviální aktivitu vytvářet, když stejného efektu bychom evidentně dosáhli i prostým přivedením výsledku výpočtu znovu na vstup podmínky, jako jsme to udělali např. v kapitole 5.10. Odpověď není jednoznačná a závisí na účelu a složitosti aplikace. U velmi jednoduchých operací bude jistě výhodnější použít přímé „zasmyčkování“ příslušné části datového toku. Oddělením spuštění bloku od dalších průběhů a současně vytvořením lineárního toku dat mezi dvěma bloky *While* naopak můžeme zjednodušit složitější algoritmus, kde by smyčky snížily čitelnost kódu. Nezanedbatelnou výhodou pak je to, že předáváním hodnot proměnné přes samostatnou aktivitu můžeme rozšířit použití cyklu přes více diagramů nebo dalších aktivit, kde by tomu jinak bránilo omezení sdílení proměnných mezi nimi.

5.12 Úloha 4 – Rekurze a možné chyby

Dalším způsobem[49] vytváření cyklů v aplikaci je použití tzv. *rekurze*. To je postup známý ze všech běžných vyšších programovacích jazyků, kdy v těle metody (funkce) voláme tutéž metodu (funkci) a takto postupně zanořujeme další volání, až do splnění nějaké podmínky. Následně se pak v opačném pořadí vracejí návratové hodnoty.

Tuto možnost nabízí také VPL a je demonstrována ve většině prací, které se VPL zabývají (samozřejmě dokumentace MRDS, dále pak [21, 1] a další). V následující úloze si ale ukážeme, kde můžeme při použití rekurze také udělat chybu.

5.12.1 Zadání

Vytvořit časovač, který bude každou sekundu inkrementovat stav svého počítadla a po dosažení 5 s upozorní uživatele. Implementace za použití běžného cyklu i rekurze.

Poznámka 5.24 Je zjevné, že tohoto efektu bychom asi dosáhli pouhým použitím jednoho timeru se zpožděním nastaveným na 5 s. Zde nám ale jde o demonstraci algoritmických postupů, takže budeme předstírat, že tato úloha má praktické využití.

5.12.2 Postup řešení

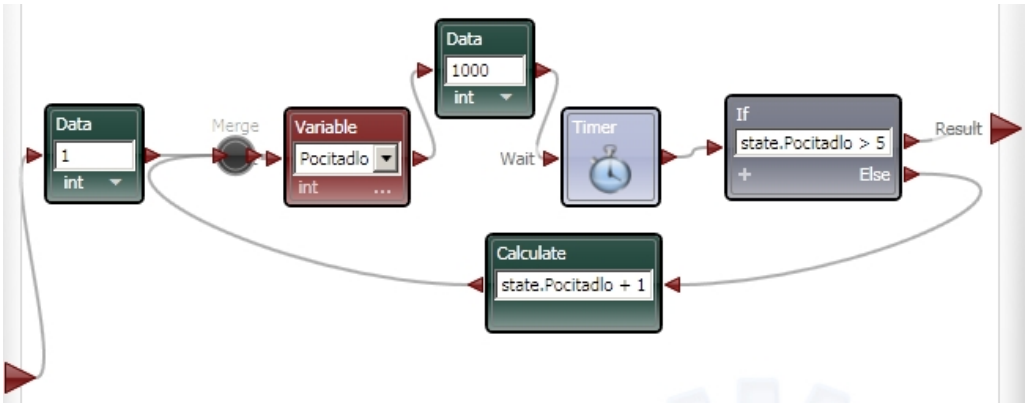
- **Bez rekurze**
  Vlastní algoritmus bez použití rekurze by po absolvování předchozích kapitol neměl být vážným problémem. Pro snadnou demonstraci různých postupů si pro tento účel vytvoříme samostatnou aktivitu nazvanou *Bez_rekurze*. Ta bude obsahovat jedinou akci, jejíž kód vidíme na obr. 19.

  V hlavním diagramu si vytvoříme již známou spouštěcí proceduru a funkci můžeme odzkoušet.

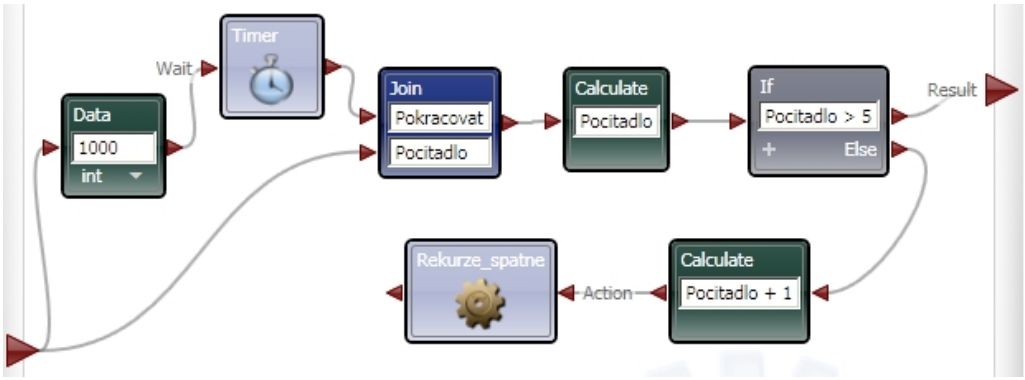
- **Rekurze**
  Nyní se pokusíme využít předností rekurze. Algoritmus je navržen tak, že dokud neuplyne stanovený počet opakování (nastavený čas 5 s), budeme stále volat vlastní aktivitu a předávat jí aktuální hodnotu počítadlo. Aktivita se tedy bude chovat stejně, jako by byla zavolána přímo z hlavního diagramu, ovšem se stále vzrůstající hodnotou počítadla. Protože současně v každém průchodu testujeme hodnotu tohoto počítadla, nemůže se stát, že vytvoříme tzv. nekonečnou smyčku (infinity loop) a program „zamrzne“. Stačí nám nastavit, že po dosažení mezní hodnoty počítadla vyšleme jeho hodnotu na návratový port. Současně můžeme eliminovat použití stavových proměnných, které už mezi jednotlivými průchody nemusíme uchovávat, hodnota počítadla bude totiž předávána jako parametr při volání aktivity. Takto vytvořenou aktivitu vidíme na obr. 20.

  Už z názvu aktivity je patrné, že asi nebude úplně v pořádku, když ji spustíme, výsledku se nedočkáme. Abychom chybu odhalili, je třeba si postupně projít jednotlivé

[49] A tvůrci MRDS (viz [21]) preferovaným, protože nevyžaduje sdílení proměnných.



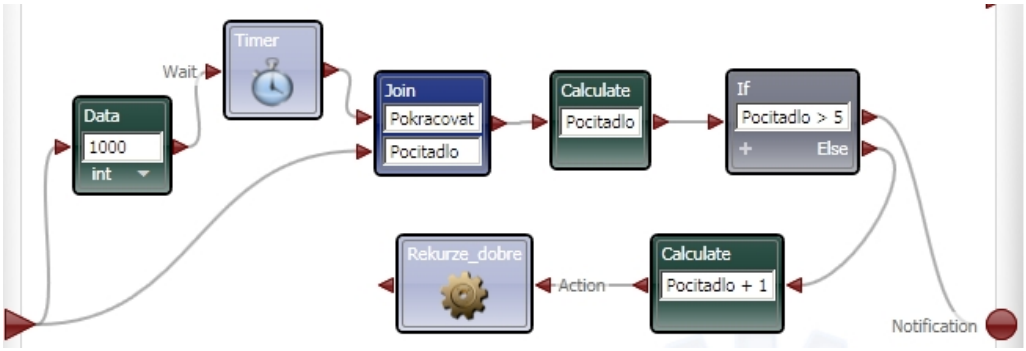
Obrázek 19: VPL_Rekurze – aktivita *Bez_rekurze*



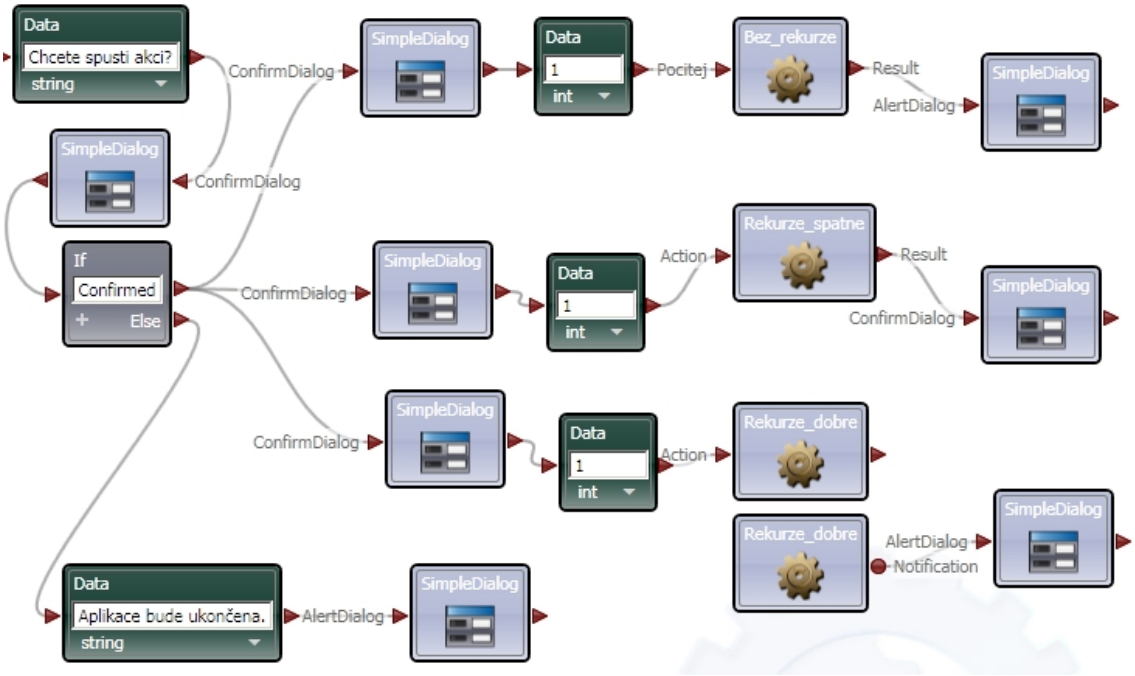
Obrázek 20: VPL_Rekurze – aktivita *Rekurze_spatne*

fáze vykonávání tohoto kódu. Tak zjistíme, že po prvním spuštění aktivity z hlavního diagramu čekáme na jejím výstupním portu na odpověď. Ve vlastní aktivitě ovšem v prvním průchodu odpověď nevyšleme, protože počítadlo je pod mezní úrovní, proto voláme znovu tutéž aktivitu. Tudíž aktivita postupně volá sama sebe a inkrementuje počítadlo, až v nejnižší úrovni, kdy počítadlo dosáhne stanovené meze, vyšle odpověď na svůj výstupní port. Tato návratová hodnota by měla být předána jako odpověď na volání z úrovně o jednu vyšší, ale protože na výstupním portu není navázán žádný další blok, datový tok se zde ukončí a uživatel se o dokončení akce nemá jak dozvědět.

- **Rekurze podruhé**
  Tuto chybu bychom mohli napravit tím, že výstupní port volané aktivity přivedeme přes *Merge* na výstup volající aktivity, čímž je zajištěno, že odpověď dorazí o úroveň výše v každém případě. Toto řešení sice bude fungovat, ale protože v bloku *Merge* slučujeme dvě větve z rozdílných výstupních bloků, nemůžeme takto jednoduše předat zpět hodnotu počítadla. Pokud ji nepotřebujeme, můžeme toto řešení použít.



Obrázek 21: VPL_Rekurze – aktivita *Rekurze_dobre*



Obrázek 22: VPL_Rekurze – výsledný diagram

Daleko čistějším řešením je ale použití notifikačních portů. Místo odpovědí na výstupním portu stačí v okamžiku překročení počítadla vyvolat odpovídající notifikaci a tu přijímat přímo v hlavním diagramu. Tím zajistíme, že na notifikační port budeme zasílat vždy pouze jeden typ zprávy (aktuální stav počítadla) a to pouze v okamžiku, kdy je skutečně hodnota počítadla překročena. Takovéto řešení vidíme na obr. 21. Celkový pohled na diagram se všemi třemi způsoby řešení pak na obr. 22.

6 LEGO® MINDSTORMS® NXT

Dánská firma LEGO® se od svého založení v roce 1932 zabývá výrobou a prodejem hraček. Celosvětově známou se stala především svými stavebnicemi, jejichž rozmanitost se stále zvyšuje, ale základem zůstává tradiční LEGO® brick (kostka LEGO®).

V roce 1998 uvedla na trh první stavebnici mikropočítačem řízeného robota a nazvala ji MINDSTORMS® Robotics Invention System (dále RIS). Skládala se z programovatelné kostky RCX osazené osmibitovým mikroprocesorem, několika senzorů a motorů a množství spojovacího materiálu. Z takovéto stavebnice bylo možno konstruovat velké množství pohyblivých a autonomních robotů. Samotný název MINDSTORMS pochází z titulu knihy Seymoura Paperta, která pojednávala o využití počítačů ve výuce dětí. Ne jen k jejich zkoušení, ale i k vlastnímu bádání a řešení problémů. Stavebnice tak završila dlouholetou spolupráci firmy LEGO® a Massachusetts Institute of Technology (MIT), na jehož půdě se Seymour Papert výzkumem zabýval [22].

Na celosvětový úspěch tohoto produktu firma LEGO® navázala v roce 2006 uvedením nové řady LEGO® MINDSTORMS® NXT. Jednalo se o stavebnici založenou na stejné koncepci jako RIS, ale s mnoha významnými inovacemi. Zejména původní 8 bitový mikroprocesor byl nahrazen 32 bitovým a Bluetooth nahradil původní infračervené připojení.

V létě roku 2009 pak přišla na trh aktualizovaná verze LEGO® MINDSTORMS® NXT 2.0 (kódové označení 8547). Ta již přinesla pouze drobnější vylepšení, jejichž detailní soupis nalezne zájemce na webu výrobce.

6.1 Popis stavebnice

V době psaní této práce měl autor k dispozici LEGO® MINDSTORMS® NXT v původní verzi z roku 2006 s originálním FW a na této verzi také všechny dále uváděné příklady testoval.

6.2 Popis a funkce jednotlivých prvků

„Mozkem“ všech NXT robotů je Kostka, jejíž mikroprocesor se stará o provádění programového kódu, komunikaci se senzory a motory a také přes USB nebo Bluetooth s připojeným PC.

O komunikaci s okolním prostředím se starají senzory, které jsou připojeny kablíkem ke vstupům Kostky. Jejich umístění v konstrukci robota je dáno významem jimi sledovaných veličin pro funkci robota. Například ultrazvukový senzor nahrazuje robotovi nejčastěji jeho „oči“ a bývá proto zpravidla umístěn ve směru pohybu.

Pohyb celého robota nebo jeho částí umožňují krokové motorky,[50] které jsou rovněž v počtu tří kusů součástí stavebnice.

[50] Na rozdíl od běžných motorů je pohyb krokového motoru řízen sérií pulsů, z nichž každý posune rotor o přesně stanovený a vždy stejný úhel. To umožňuje přesné řízení např. ujeté vzdálenosti, úhlu otočení kloubu apod.

Obrázek 23: Kostka NXT s připojenými senzory a motory

Obrázek 24: Detail Kostky NXT

Robot může obsluhu informovat prostřednictvím programově ovládaného displeje Kostky nebo akustickým signálem integrovaného reproduktoru. Naopak obsluha může zasáhnout do běhu programu tlačítky na Kostce.

6.2.1 Kostka

V originální dokumentaci i v anglicky psaných zdrojích je hlavní jednotka NXT nazývána Brick (cihla). V češtině se ovšem už od počátku stavebnic LEGO® na našem trhu ujal volný překlad kostka, který rovněž uvádí česká dokumentace ke stavebnici LEGO® MINDSTORMS® NXT. Proto jej budeme používat i zde, pro snadnou orientaci v textu a pro odlišení od obyčejných prvků stavebnice s velkým počátečním písmenem, tedy Kostka.

Z důvodu snadné výměny baterií bývá Kostka umístěna na takovém místě robota, které umožňuje její snadné vyjmutí. Toto umístění bývá výhodné i pro snadný přístup k ovládacím tlačítkům a displeji Kostky.

Obrázek 25: Dotykový senzor (bumper)

6.2.2 Senzory

Senzory slouží robotu k získání informací o objektech reálného světa, ve kterém se pohybuje, a jsou tedy nezbytné pro plnění jakékoli úlohy, která není čistě statická a vyžaduje nějakou interakci s okolím. Senzory tedy vlastně nahrazují lidské smysly a dokonce je mohou i překonat jak druhy zpracovávaných informací, tak přesností jejich vyhodnocení. Typ a kvalita použitých senzorů do značné míry ovlivňuje užitnou hodnotu robota a na rozdíl od čistě SW aplikací ji většinou nelze nahradit [51] třeba použitím efektivnějších algoritmů nebo vyšší rychlostí procesoru. Je zřejmé, že existuje přímá úměra mezi kvalitou a cenou senzoru, proto od těch, které nalezneme ve stavebnici LEGO® MINDSTORMS® NXT, nemůžeme očekávat parametry vhodné pro profesionální nasazení. Pro naše účely, tedy zejména první seznámení s robotickými aplikacemi a jejich výuku, ovšem plně postačují. Výrobce zvolil takovou kombinaci druhů snímaných veličin, počtu příslušných senzorů a kvalitu jejich zpracování, aby bylo možné prezentovat všechny základní problémy, které musí robot v terénu řešit.

Jednotlivé senzory si stručně popíšeme a uvedeme i modelové situace, ve kterých se uplatní. Pro detailnější seznámení zejména s technickými parametry, interním zapojením a změřenými charakteristikami můžeme odkázat na [23, 24, 25].

6.2.3 Dotykový senzor

Zprostředkovává informaci o fyzickém kontaktu robota s okolím, např. náraz do překážky, uchopení předmětu, povel obsluhy (simulace stisknutí tlačítka) apod.

Jedná se o dvoustavový spínač (sepnuto / nesepnuto) s pevně nastavenou hodnotou tlaku nutného pro sepnutí a jde tedy vlastně pouze indikátor stavu. Jeho výhodou je okamžitá[52] a přesná reakce na událost bez nutnosti kalibrace, naopak nevýhodou je

[51] Za jistých okolností lze důmyslnými algoritmy zlepšit kvalitu získávaných informací např. kombinací údajů z více senzorů, eliminací šumů v datech apod., případně i ze dvou veličin vypočítat veličinu třetí a nahradit tak samostatný senzor.

[52] Někdy ovšem může jít o nevýhodu. Při tlaku, který je vyvíjen na senzor a který je blízký jeho prahové hodnotě sepnutí, může docházet k zakmitání, kdy senzor vyšle postupně sled opačných stavů. To může působit problémy zejména v aplikacích, které na základě této informace vyhodnocují počet kontaktů, např.

Obrázek 26: Optický senzor

nutnost „strategického“ umístění na těle robota, aby byla zajištěna reakce na všechny podněty, které se mohou vyskytnout. Ve většině případů je nezbytné instalovat větší počet těchto senzorů, nebo je kombinovat s jinými typy, např. pro snímání vzdálenosti a směru k překážce.

6.2.4 Světelný (optický) senzor

Tento senzor snímá hodnotu intenzity osvětlení a lze jej tedy využít k orientaci v prostoru nebo předávání pokynů prostřednictvím světelných signálů. Je ovšem rovněž vybaven červenou LED, která umožňuje snímat intenzitu světla odraženého od předmětu, což bývá nejčastější případ aplikace. Robot se čidlem pohybuje v těsné blízkosti zkoumaného objektu (podložka, překážka apod.) a podle množství odraženého světla vyhodnocuje jeho barvu (tedy spíše poměr odraženého a pohlceného světla) nebo vzdálenost.

6.2.5 Zvukový senzor (mikrofon)

Jedná se o jednoduchý mikrofon, který podává robotu informaci o akustickém tlaku. Výsledná hodnota je interpretována v procentech úrovně 90 dB. Prostřednictvím tohoto čidla může robot buď prozkoumávat terén (orientovat se podle hluku v prostředí) nebo častěji spíš reagovat na povely a pokyny obsluhy. V testovacích aplikacích lze takto například zařadit nouzové zastavení, kdy hlasitý povel obsluhy vyvolá provedení vhodné části programového kódu.

6.2.6 Ultrazvukový senzor (sonar)

Je založen na principu měření doby šíření ultrazvukového signálu od robota k překážce a zpět. Pracuje v pulzním režimu,[53] tzn., že informace o vzdálenosti nejsou spojité, ale jsou

sledují počet stisknutí tlačítka, vytříděných mincí apod. S tím je třeba počítat při návrhu aplikace a zjišťovat stav senzoru vícekrát a reagovat pouze při shodném výsledku několika měření. Podobný problém může způsobit i příliš rychlá změna stavu senzoru, která je kratší než perioda s jakou kontroluje stav senzoru aplikace. V dalším textu (kapitola 7.23.2) si ukážeme, že právě toto může lépe řešit aplikace řízená událostmi.

[53] Jedná se o jediné čidlo ve standardním balení stavebnice, které využívá digitální sběrnici Kostky [23].

Obrázek 27: Zvukový senzor

Obrázek 28: Ultrazvukový senzor (sonar)

vzorkovány a předávány v digitální podobě. Údaj sonaru je interpretován jako vzdálenost předmětu v cm, přičemž maximální vzdálenost je 255 cm. Princip sonaru ovšem vyžaduje kvalitní odraz zvuku od zkoumaného předmětu, tzn. jak jeho vhodný povrch, tak úhel, pod kterým se vlna odráží. Přesnost bývá přibližně 3 cm [25], ale ve většině případů je třeba sonar pro konkrétní aplikaci zkalibrovat. Důležité je také mít na zřeteli poměrně široký 30° úhel, pod kterým je zvuková vlna vysílána, nelze tedy zajistit maximální směrovost.

6.2.7 Barevný optický senzor

Od verze NXT 2.0 (v prodeji od srpna 2009) se ve stavebnici LEGO® MINDSTORMS® NXT dodává další typ senzoru, který byl dříve dostupný pouze jako samostatně prodávaný doplněk. Jde o barevný optický senzor (obr. 29) a je založen na podobném principu jako původní optický senzor, jen je zde pro přisvětlení použito tří různobarevných LED a je vyhodnocováno množství odraženého světla pro každou barvu zvlášť. Tím je možné zjistit nikoli pouze kontrast, ale i barvu zkoumaného povrchu. Stejně jako optický senzor je ale citlivý na šum v podobě pronikajícího okolního světla.

Obrázek 29: Barevný optický senzor

Autor měl při psaní tohoto textu k dispozici pouze základní sadu senzorů z první verze stavebnice (označení 8527), proto jsou všechny následující úlohy demonstrovány na nich.

6.2.8 Další senzory od třetích stran

Postupem času se na trhu objevují nové typy senzorů od nezávislých výrobců (např. HiTechnic, Mindsensors), které jsou s LEGO® MINDSTORMS® NXT kompatibilní. Máme tedy možnost použít např. víceosý akcelerometr, který lze výhodně použít třeba pro udržení stability balancujícího robota, kompas pro orientaci podle magnetického pólu Země, infračervený detektor pro detekci tepelných zdrojů apod.

6.2.9 Servomotory

Pohyb robota zajišťují servomotory, které rovněž ve stavebnici najdeme. Jejich počet a způsob použití záleží na modelu robota, který hodláme postavit, proto jsou ve stavebnici k dispozici celkem tři (Ke Kostce lze připojit současně nejvýše čtyři motory). V dalších úlohách budeme používat výhradně model robota *Tribot*, který je vybaven dvěma motory na přední nápravě (každé kolo pohání vlastní motor) a třetí motor slouží k ovládání manipulačních čelistí v přední části robota.

Motory jsou řízeny pulzní šířkovou modulací, tzn., že podle požadované rychlosti otáčení je měněna šířka (délka) impulzu a tím množství energie, které se může přeměnit na pohyb.

Každý motor navíc obsahuje integrovaný detektor otáček (na podobném principu jako u počítačové myši), což umožňuje přesné řízení v toleranci 1°. Velmi podrobné vysvětlení interního zapojení a řízení motorů lze nalézt v [23].

6.2.10 Baterie

Napájení Kostky a jejím prostřednictvím i všech periferií je řešeno pomocí šesti tužkových baterií (typ AA), které jsou vloženy přímo v těle Kostky. To s sebou nese problémy při

Obrázek 30: Servomotor

Obrázek 31: Přídavný battery pack

výkonově náročnějších aplikacích, protože spotřeba energie zejména motory je opravdu značná a měnit baterie je potřeba velmi často. Proto je vhodné Kostku vždy na těle robota umisťovat tak, aby byla snadno demontovatelná.

Další možností je pořízení samostatně dodávaného *Battery packu* od firmy LEGO®, který obsahuje akumulátory a lze jej jako celek dobíjet (obr. 31). Bohužel rozměrově o několik milimetrů přesahuje původní rozměry Kostky (umisťuje se do zadní části Kostky, kde původně byly baterie), proto u některých modelů musíme sáhnout ke změně konstrukce.

6.2.11 Komunikační rozhraní

Kostka LEGO® MINDSTORMS® NXTje vybavena dvěma rozhraními pro komunikaci s PC. Prvním je USB, které ovšem nemůžeme využít při programování robota z MRDS, druhým je pak Bluetooth.

Bluetooth (dále BT) je preferovaná volba pro komunikaci mezi PC a robotem, umožňuje nám robota řídit bezdrátově i ve volném terénu. Samotné spojení s robotem je podobné jako u běžných BT zařízení. Nejdůležitější je správná instalace BT adaptéru

na PC.[54] Postup instalace (ačkoliv není nijak složitý) je detailně vysvětlen na stránce výrobce [26], včetně řešení nejčastějších problémů. Následně pouze na Kostce v jejím menu vybereme volbu Bluetooth a povolíme režim vyhledávání (discoverable). V dalším kroku na PC vyhledáme všechna dostupná BT zařízení v okolí a mezi nimi bychom měli nalézt také robota. Toho ze seznamu nalezených zařízení vybereme a potvrdíme požadavek na připojení. V tomto okamžiku nám Kostka nabídne k potvrzení (změně) heslo, kterým se PC musí prokázat (výchozí je 1234). Toto heslo zadáme do příslušného pole v PC a akci dokončíme. Nyní by se měla mezi nainstalovaným BT HW objevit i příslušná ikona našeho robota (konkrétní zobrazení záleží na použitém operačním systému a ovladačích BT).

Poznámka 6.1 Pro další práci budeme nutně potřebovat znát číslo sériového portu, který je pro BT spojení emulován. To nalezneme ve vlastnostech právě vytvořeného připojení.

Při vlastním programování aplikací a jejich testování na reálném robotovi nám správnou funkci BT komunikace potvrdí krátké pípnutí po spuštění aplikace a současně zobrazení webového rozhraní DSS služby Kostky, kde je stav spojení rovněž zobrazen (zobrazování webu lze nicméně v konfiguraci služby vypnout).

6.2.12 Ostatní příslušenství

Protože se jedná opravdu o stavebnici, najdeme v balení mimo výše zmíněných aktivních prvků také stovky dalších dílů, které známe z běžných LEGO® stavebnic. Jde o různé osky, spojovací dílky, kolečka, převody apod. Přesný výčet je uveden v příbalovém letáku nebo na webu výrobce a prodejců. Pro nás je podstatné, že obsahuje všechny díly nutné k sestavení dále popisovaného modelu robota Tribot a také všech modelů, které nalezneme na stránkách výrobce. Všechny díly jsou, jak je u stavebnic určených dětem obvyklé, barevně odlišeny a detailně vyobrazeny v přiloženém návodu, takže jejich správná identifikace podle konkrétního stavebního návodu je velmi snadná.

6.2.13 Try Me

Originální FW kostky obsahuje velmi užitečnou funkci Try Me. Po jejím spuštění je možné ověřit stav a funkci všech připojených senzorů a motorů. Snadno tak odhalíme špatně zapojený kabel nebo vadný senzor ještě před sestavením celého robota a odpadne tak jeho případná demontáž. Tuto funkci můžeme použít i v případě, kdy naše aplikace nefunguje podle představ a máme podezření na HW závadu.

[54] Bohužel ne všechny BT adaptéry jsou s robotem LEGO® MINDSTORMS® NXTplně kompatibilní. Autor měl možnost vyzkoušet tři adaptéry různých výrobců (dva integrované v notebooku a jeden externí) a ve všech třech případech bylo spojení bez problémů. Přesto lze na internetu nalézt celou řadu diskuzí, ve kterých se řeší právě kompatibilita BT adaptérů. Přímo společnost LEGO® dodává vlastní BT adaptér, u kterého funkčnost garantuje.

6.3 Nativní vývojové prostředí NXT-G

Stavebnice je primárně určena dětem (dle výrobce od 10 let), u kterých se nepředpokládá předchozí zkušenost s programováním. Proto je součástí dodávky také speciální programovací prostředí *NXT-G* vyvinuté ve spolupráci LEGO® Education a National Instruments. Jde o optimalizovanou verzi profesionálního grafického programovacího prostředí LabVIEW™ pro účely stavebnice a obsahuje předdefinovanou sadu prvků (zde nazývaných bloky), jejichž skládáním se sestavuje algoritmus aplikace.

Tyto bloky lze dále rozšiřovat a mezi internetovou komunitou nadšenců se stále objevují nové a aktualizované bloky pro různé účely. Samotný tvůrce aplikace může bloky vytvářet přímo ve vývojovém prostředí kombinací původních (jde vlastně o jakési podprogramy), nebo je vytvořit v originálním prostředí LabVIEW™. Jde o hlavní a nejrozšířenější způsob programování robota, proto se mu věnuje celá řada publikací, za všechny jmenujme např. [27]. Velmi dobrý multimediální kurz jazyka může zájemce nalézt online na [28].

6.4 Alternativní FW a jazyky

Postupem času, s postupným rozšiřováním stavebnic, se začalo objevovat širší spektrum jazyků a prostředí, ve kterých je možné aplikace pro LEGO® MINDSTORMS® NXT vyvíjet. Pro některé z nich je třeba do Kostky nahrát upravený firmware, některé využívají ten stávající. Přehled některých z těchto jazyků najde zájemce například v [2]. Celá kniha věnovaná programování robota v Javě je pak [29], byť se zabývá především předchozí verzí RCX.

Internetová komunita také vyvíjí prostředky pro HW ladění aplikací přímo v zařízení (in-device debugging), viz např. [30].

Pro nás je podstatné, že již s první verzí MRDS lze roboty programovat i v prostředí .NET a navíc i vizuálním jazykem VPL.

6.5 Internetová komunita

Jako u každé moderní technologie, i v tomto případě nalezne zájemce nejčerstvější a nejúplnější informace na internetu. Autor může doporučit třeba web [31] bohatě zásobený stavebními návody pro různé modely robotů, případně [32]. Samozřejmě přímo od zdroje jsou informace na webu výrobce [33].

6.6 Model Tribot

V našich úlohách budeme využívat výhradně dvě modifikace robota nazvaného Tribot. Jde o základní model, který je k dispozici i pro simulaci ve virtuálním prostředí (jak uvidíme dále) a současně je velmi snadné jej sestavit. Stavební plánek je obsažen již v přibalené příručce ve stavebnici a stavba včetně oživení by neměla trvat déle než dvacet minut.

Obě varianty robota se liší pouze třetím motorem s připojenými čelistmi. Mimo úlohy v kapitole 7.18, ve které budeme zkoušet právě ovládání tohoto motoru, je tedy lhostejné, kterou z variant si postavíme.

V každém případě bychom si ale po sestavení robota měli ověřit zapojení aktivních prvků a jejich správnou činnost programem *Try Me*, který najdeme v menu Kostky. Současně by bylo vhodné si někam poznamenat, kam jsou které senzory a motory na Kostce připojeny. To nám může ušetřit mnoho času při hledání podivných závad.

7 Vývoj aplikací pro LEGO® MINDSTORMS® NXT v MRDS

7.1 Použité typy robotů

Účelem tohoto textu je demonstrovat vývoj robotických aplikací obecně, proto se také budeme snažit o co nejobecnější postupy. V rámci předvedení výsledků však musíme zvolit takovou platformu, kterou máme k dispozici a o které lze předpokládat, že jí bude disponovat i potenciální čtenář. Zde tedy, v souladu se zadáním, volíme robota sestaveného ze stavebnice LEGO® MINDSTORMS® NXT. Pro většinu aplikací se ovšem budeme snažit o využití virtuálního prostředí MRDS a tomu přizpůsobíme i výběr vhodného modelu robota. Jako nejuniverzálnější se jeví tzv. TriBot, což je jeden ze základních modelů popsaných v příručce ke stavebnici a současně model podporovaný v obou dále zmíněných virtuálních prostředích.

7.2 Generické služby

Protože obecně požadujeme tvorbu aplikací pro nejrůznější modely robotů, je nutné, aby námi vytvořená aplikace dokázala s daným hardwarem správně komunikovat. Při předpokládané šíři nasazení by to vyžadovalo, abychom měli vždy k dispozici speciální služby (pokud možno dodané výrobcem HW), které bychom do své aplikace integrovali. To by ovšem mělo zásadní dopad na přenositelnost již jednou vytvořených aplikací na jiné HW platformy. V neposlední řadě by pak autoři aplikací museli mít přístup i ke konkrétnímu cílovému HW, aby měli možnost kód odladit.

Tvůrci MRDS proto už do základu vývojového studia vložili základní prvek přenositelnosti kódu – generické služby. Jde o jakési univerzální soubory operací a zpráv, které slouží k ovládání nějaké obecně definované části HW robota. Na výrobci pak pouze je, aby tyto univerzální operace implementoval do svého HW a dodal popis (tzv. manifest), kterým řekne DSS uzlu, pod kterým aplikace běží, které (tentokrát již HW specifické) služby má místo oněch generických použít. Pokud tedy programujeme aplikaci pro roboty s podobným designem po stránce řízení a vybavení, máme možnost vytvořit jednu univerzální aplikaci, se kterou pak pouze dodáme příslušný manifest.

Konkrétní použití manifestů si ukážeme nejlépe současně s vývojem praktických VPL aplikací na konkrétních příkladech.

V základní nabídce služeb máme k dispozici mimo jiné diferenciální pohon robota[55], obecný analogový senzor, či obecný tlakový detektor.[56] Tyto tři budeme v dalších úlohách využívat nejčastěji.

[55] Jde o variantu řízení, která využívá pro pohon každého kola na téže nápravě samostatný motor. Pohyb do stran je pak řešen změnou otáček obou motorů a z jejich rozdílu (odtud diferenciální) pak vznikne výsledný směrový vektor pohybu robota. Principiálně jde o stejný systém, jaký používají třeba tanky.

[56] Jde o běžný spínač se dvěma stavy: sepnuto / nesepnuto.

7.3 HW specifické služby

Součástí MRDS je i balík specifických služeb pro některé rozšířenější typy robotů a jejich vybavení. My budeme dále využívat pouze služby pro LEGO® MINDSTORMS® NXT, a to především ty, ke kterým neexistuje ekvivalentní náhrada v podobě generických služeb.

7.4 Robotické VPL aplikace

V tomto okamžiku již máme za sebou seznámení se základy jazyka VPL (kapitola 5), známe možnosti a složení stavebnice LEGO® MINDSTORMS® NXT a také již víme, jaké možnosti nám MRDS nabízí. Můžeme tedy konečně přistoupit k tvorbě skutečných robotických aplikací v jazyce VPL.

Následující úlohy jsou sestaveny tak, aby demonstrovaly pokud možno co nejširší paletu možností a vlastností jazyka VPL a prostředí MRDS ve vztahu k programování robotů. Zdaleka nejde o výčet úplný, některé postupy lze modifikovat, případně řešit zcela jinak, ovšem vodítkem zde byla snaha o vytvoření příkladů typických, nikoli optimálních. Ačkoli jsou úlohy na sobě nezávislé, předpokládá se jejich studium v pořadí, v jakém jsou uvedeny, neboť většina nových pojmů je vysvětlena pouze jedenkrát, při jejich prvním použití, a následně se již počítá s jejich znalostí. Tento postup také dovoluje částečně využívat již jednou vytvořené části kódu.

Řešení úloh je členěno na dílčí kroky, které představují jednotlivé typické fáze tvorby aplikace (analýza, použité služby, sestavení diagramu, spuštění). V případech, kdy to povaha problému vyžaduje, jsou pak doplněny kroky pro danou úlohu specifickými.

7.5 Úloha 5 – Čelem vzad

7.5.1 Zadání

První praktickou úlohou bude otočit robota o 180 stupňů, tedy vykonat povel „čelem vzad“. Abychom dostáli co nejobecnějšímu přístupu, budeme v této fázi používat pouze generické služby, které nám umožní později změnou manifestů zvolit cílovou platformu pro prezentaci výsledku.

7.5.2 Postup řešení:

- **Analýza**
  Tato úloha sestává pouze z jedné operace robota. Měl by provést jednu otočku o 180°, takže algoritmus nebude nikterak složitý a jádro problému tkví ve výběru vhodných služeb, které nám umožní robota ovládat.

- **Výběr služeb**
  Nejprve si tedy musíme rozmyslet, jaké služby budeme ke splnění úkolu potřebovat. Protože budeme v našich úlohách používat vždy model *Tribot*, stačí nám zjistit, jaké prvky využívá k pohybu. Zjistíme, že je osazen pouze dvěma motory pro pohon kol na přední ose, zadní kolečko poháněné není. Dále je osazen dotykovým čidlem (bumper), sonarem[57] a světelným senzorem.

  Robot má provádět pouze pohyb, který není nijak závislý na vnějším prostředí, nebudeme tedy potřebovat žádná čidla ani senzory. Můžeme tedy použít generickou službu pro obsluhu diferenciálního pohonu *Generic Differential Drive* (dále jen GDD). Dále budeme stejně jako v každé další úloze používat dialog pro spuštění akce, proto přidáme i *Simple Dialog*.

- **Sestavení diagramu**
  Nyní bychom měli nahlédnout do nápovědy MRDS a vyhledat popis GDD, abychom zjistili, jaké zprávy služba přijímá a jaké operace je schopna provést. Náhradní možností (ale ne rovnocennou) je zkusmo provést spojení obou dosud vložených bloků. Editor nám nabídne seznam zpráv, které můžeme GDD odeslat. Z nich podle názvu odvodíme, že asi nejvhodnější po naše účely bude *RotateDegrees*. Pokud tuto volbu potvrdíme, zobrazí se nabídka parametrů, které můžeme službě ve zprávě poslat. Opět není těžké odvodit, že požadovaný úhel předáme jako *Degrees* a rychlost pohybu *Power*. Podržíme-li myš nad názvy těchto parametrů, zjistíme, že hodnota úhlu se zadává jako číslo typu *double* ve stupních a že výkon může nabývat hodnoty od -1 (naplno vzad) po +1 (naplno vpřed). Poslední parametr udává stav motoru, a jak bychom zjistili v dokumentaci, vyplňovat ho nemusíme.

  Pokud tedy máme kompletní vstupní část s dialogem (tu už známe z předchozích úloh), můžeme zadat hodnoty přímo do zprávy mezi dialogem a GDD. Použijeme tedy volbu *Edit values directly* a vyplníme 180 a například 0.7.

[57] Dále budeme používat pro ultrazvukový senzor měřící vzdálenost termín sonar a pro dotykové (nárazové) čidlo termín bumper (nárazník).

Poznámka 7.1 Tato metoda tzv. „natvrdo“ zadávaných dat sice výrazně zpřehlední výsledný diagram (pokud jde o zaplnění bloky) a také ušetří čas při návrhu, ale na druhé straně je takovýto kód pro nezasvěceného (a tím se po čase ve většině případů stane i sám autor aplikace) naprosto nečitelný, předávané hodnoty nejsou viditelné.

- **Spuštění**
  Nyní se pokusíme aplikaci spustit. Úvodní dialog se sice objeví, ale současně uvidíme v konzoli DSS uzlu mezi výpisem spouštěných služeb i červené chybové hlášky, které oznamují: *Partner enumeration during service startup failed* a v dalším textu objevíme, že chyba pochází od služby GDD. Tím nám dal DSS uzel, který služby spouští, najevo, že pro generickou službu nenalezl její skutečnou implementaci pro nějaký HW.

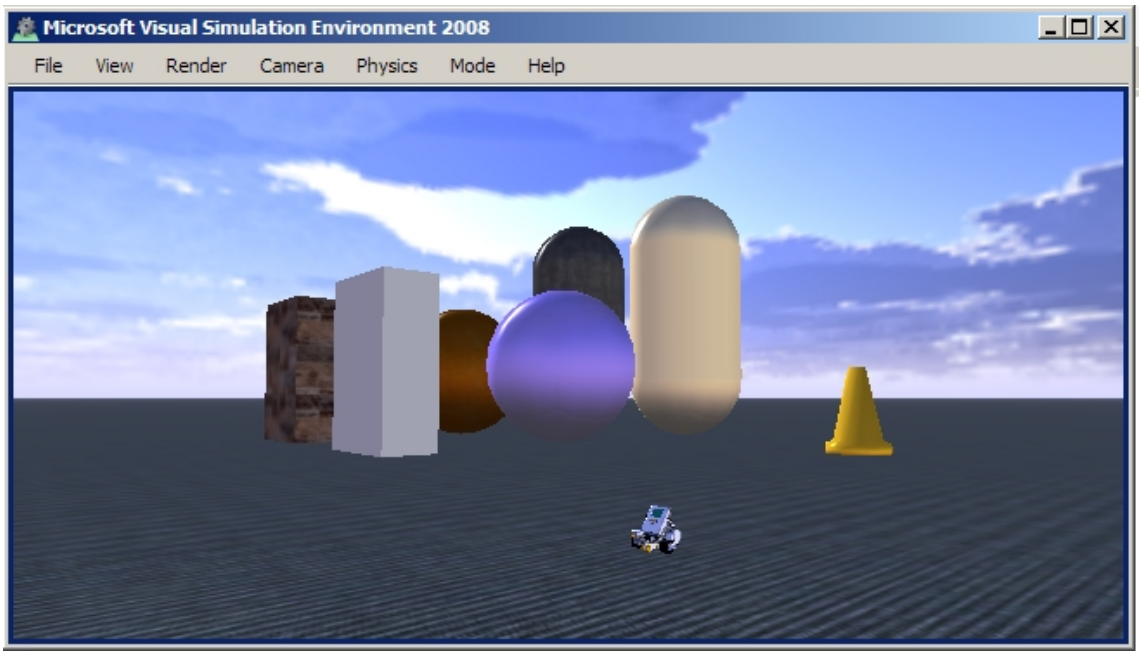
- **Úpravy**
  Nyní trošku předběhneme kapitolu a pokusíme se problém vyřešit. Pokud klepneme na GDD, můžeme si v pravém okně editoru všimnout, že máme možnost nastavit konfiguraci služby. Z rolety tedy vybereme v tomto případě položku *Use a manifest* a následně *Import*. Měl by se nám zobrazit seznam všech kompatibilních manifestů, které MRDS pro naši službu našlo. Z tohoto seznamu pro začátek vybereme *LEGO.NXT.Tribot.Simulation.Manifest.xml*.[58]

  Nyní můžeme aplikaci znovu spustit a výsledkem by měl být robot, kterého vidíme na vlastní oči opravdu provádět poloviční piruetu (viz obr. 32).

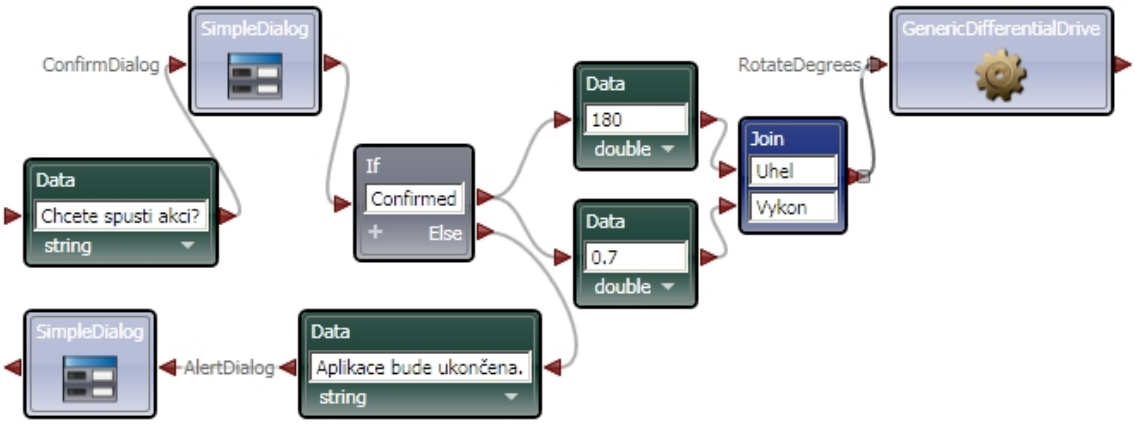
- **Dokončení**
  Toto řešení je tedy funkční, ale přece jen bychom v zájmu přehlednosti[59] měli hodnoty, které předáváme GDD vynést do samostatných bloků *Data* a teprve z nich povel odeslat. Opět si musíme pomoci blokem *Join*, abychom mohli oba parametry zařadit do jedné zprávy. Výsledný kód vidíme na obr. 33.

[58] Těchto souborů patrně v seznamu uvidíme víc, proto vybereme ten, který je umístěn v podadresáři *samples* balíku MRDS (cesta se zobrazí po přidržení ukazatele myši nad manifestem).

[59] Nicméně vidíme, že například pro rychlé otestování je možné sestavit kód i velmi jednoduše a přitom funkčně.



Obrázek 32: VPL_CelemVzad – robot ve virtuálním prostředí MRDS



Obrázek 33: VPL_CelemVzad – výsledný diagram

7.6 Možnosti využití vizualizací

7.6.1 Princip manifestů

O tom, k čemu slouží manifesty, jsme se již zmínili v kapitole 7.2. Nyní si na konkrétních příkladech ukážeme, jak můžeme velmi jednoduchou změnou manifestů nebo jejich editací dosáhnout významné změny ve výsledné aplikaci.

Manifest je vlastně XML soubor, který načítá DSS uzel při startu služby a kde nalezne další informace se službou související. Zejména jde o další služby, jejichž spuštění je vyžadováno, včetně odkazů na jejich konfigurační soubory, dále tzv. partnerské služby, což jsou reálné implementace služeb nahrazující služby generické apod.

Soubory lze editovat ručně, případně využít tzv. *DSS Manifest Editor*, který je součástí balíku MRDS, ale pro naše účely bude stačit vědět, kde manifesty najdeme a jak je použijeme.

Při instalaci balíku aplikací MRDS se v jeho instalačním adresáři [60] vytvoří podadresář *samples\config*, kde je uložena většina potřebných manifestů. To, že jde o podadresář *samples*, tedy s ukázkami, není náhoda. Samotné MRDS žádné manifesty neobsahuje, stejně jako neobsahuje žádné hotové služby pro podporu robotiky. Jde pouze o prostředí, ve kterém má programátor možnost tyto produkty vytvořit. Aby však každý nemusel začínat „na zelené louce“, vytvořili autoři MRDS i ukázkové služby včetně manifestů, které mohou dál autoři aplikací používat (samozřejmě za dodržení licenčních podmínek).

Ve výše zmíněném adresáři nalezneme i manifest, který jsme použili v předchozím úkolu. Pokud si ho prohlédneme (viz výpis 18), zjistíme, že není nijak složitý a navíc obsahuje komentáře.

```
1  <?xml version="1.0" ?>
2  <Manifest
3      xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
4      xmlns:dssp="http://schemas.microsoft.com/xw/2004/10/dssp.html"
5      xmlns:simcommon="http://schemas.microsoft.com/robotics/2006/04/simulation.html"
6      >
7      <CreateServiceList>
8
9        <!-- Start simulation engine and visualization window -->
10       <ServiceRecordType>
11         <dssp:Contract>http://schemas.microsoft.com/robotics/2006/04/simulationengine.html</
               dssp:Contract>
12         <dssp:PartnerList>
13           <dssp:Partner>
14             <dssp:Service>LEGO.NXT.Tribot.SimulationEngineState.xml</dssp:Service>
15             <dssp:Name>dssp:StateService</dssp:Name>
16           </dssp:Partner>
17         </dssp:PartnerList>
18       </ServiceRecordType>
19
20       <!-- Start simulated motor service -->
21       <ServiceRecordType>
```

[60] Nejčastěji C:\Program Files \Microsoft Robotics Dev Studio 2008 R2

```
22          <dssp:Contract>http://schemas.microsoft.com/robotics/simulation/services/2006/05/
                simulateddifferentialdrive.html</dssp:Contract>
23          <dssp:PartnerList>
24            <dssp:Partner>
25              <!--The partner name must match the entity name-->
26              <dssp:Service>http://localhost/LegoNXTMotorBase</dssp:Service>
27              <dssp:Name>simcommon:Entity</dssp:Name>
28            </dssp:Partner>
29          </dssp:PartnerList>
30        </ServiceRecordType>
31
32        <!-- Start simulated bumper service -->
33        <ServiceRecordType>
34          <dssp:Contract>http://schemas.microsoft.com/robotics/simulation/services/2006/05/
                simulatedbumper.html</dssp:Contract>
35          <dssp:PartnerList>
36            <dssp:Partner>
37              <!--The partner name must match the entity name-->
38              <dssp:Service>http://localhost/LegoNXTBumpers</dssp:Service>
39              <dssp:Name>simcommon:Entity</dssp:Name>
40            </dssp:Partner>
41          </dssp:PartnerList>
42        </ServiceRecordType>
43
44      </CreateServiceList>
45  </Manifest>
```

Výpis 18: LEGO.NXT.Tribot.Simulation.manifest.xml

Na řádce 7 požaduje spuštění dalších služeb, konkrétně *simulationengine*, která spouští virtuální prostředí, ve kterém se bude robot pohybovat, *simulateddifferentialdrive*, což je právě implementace naší generické služby GDD a *simulatedbumper*, tedy služba simulující bumper.

Služba *simulationengine* pak dále specifikuje svůj manifest, ve kterém má uloženy detaily generovaného prostředí, např. pozici kamery a jednotlivých objektů. V kapitole 7.6.6 si ukážeme, jak můžeme editací tohoto souboru změnit prostředí, ve kterém budeme robota testovat.

Další manifesty pak nalezneme v adresářích jiných aplikací, které nějakým způsobem MRDS využívají. V našem případě jde o produkt společnosti SimplySim, který přináší další implementace generických služeb pro roboty LEGO® MINDSTORMS® NXT a současně další virtuální prostředí. V podadresáři *SimplySim\NXT-MSRDS\Manifests* tak najdeme další sadu manifestů, z nichž některé budeme později potřebovat.

Dále nesmíme zapomenout na to, že editor VPL si všechny manifesty, na které se při tvorbě aplikace odkážeme, okamžitě zkopíruje do adresáře aplikace a dále pracuje pouze s těmito kopiemi. Pokud se tedy podíváme do adresáře posledního úkolu, měli bychom tam najít kopii manifestu z výpisu 18 a také všechny další, na které DSS uzel při provádění aplikace narazí.

Po nějakém čase stráveném v editoru VPL si také všimneme, že při importu manifestu jsou nabízeny všechny manifesty dostupné v podadresářích instalace MRDS, tedy včetně těch uložených společně s ukázkovými aplikace.

Poznámka 7.2 Při praktických pokusech se nám možná bude zdát manipulace s manifesty nepohodlná a nepraktická, ale musíme si uvědomit, že tím tvůrci MRDS chtěli eliminovat zásahy do již napsaného kódu, a to se jim podařilo. Při použití již zkompilované služby opravdu nemusíme znát její zdrojový kód, jen dodáme potřebné parametry prostřednictvím manifestů.

7.6.2 Základní prostředí VPL

V úloze z kapitoly 18 jsme si funkci robota ověřili v simulovaném prostředí standardně nainstalovaném s MRDS. Pokud se vrátíme k výpisu příslušného manifestu, který jsme použili (viz výpis 18), zjistíme, že toto prostředí dokáže simulovat pouze diferenciální pohon a bumper. Pro základní operace robota s nimi sice vystačíme, ale jakákoli čidla, senzory, či motory již vyzkoušet nemůžeme a museli bychom buď použít reálného robota, nebo si vytvořit simulaci vlastní. Poslední varianta ovšem připadá v úvahu pouze pro velmi rozsáhlý projekt, kdy se nám investice do tak složité činnosti, jako je simulace fyzikálních vlastností reálného objektu, vyplatí.

7.6.3 Prostředí SimplySim

Naštěstí s rozšířením MRDS mezi vývojáře se objevují i volně přístupné simulace třetích stran, které můžeme jednoduše využít. My se zaměříme na velmi povedené prostředí společnosti SimplySim, které obsahuje simulaci téměř kompletní[61] sady senzorů dodávaných ve stavebnici LEGO® MINDSTORMS® NXT a navíc prostředí samotné má mnohem blíž realitě než výchozí, skutečně syntetické, prostředí MRDS.

Pokusme se tedy naši úlohu vyzkoušet v tomto novém prostředí. V tuto chvíli bychom již měli mít aplikaci SimplySim *NXT-MSRDS* nainstalovánu,[62] proto bychom měli být schopni stejným postupem jako v kapitole 7.5.2 importovat správný manifest nazvaný *SimulationStandalone.Manifest.xml*. Tentokrát ho budeme hledat v podadresáři *SimplySim\NXT-MSRDS\Manifests* instalace MRDS.

Zkusme si nyní aplikaci znovu spustit. Nové prostředí i s robotem je na obr. 34. Je zde sice použit mírně odlišný model robota Tribot, je navíc vybaven světelným senzorem, sonarem a motorem ovládajícím čelisti, ale podvozek a bumper jsou shodné, takže můžeme aplikace, které využívají pouze těchto služeb, simulovat v obou prostředích.

[61] Chybí pouze zvukový senzor.

[62] V opačném případě si instalační soubor můžeme stáhnout z [9] a instalaci podle pokynů z kapitoly 2.9 provést nyní.

Obrázek 34: VPL_CelemVzad – robot ve virtuálním prostředí SimplySim

7.6.4 Aplikace SimplySim

Další možností, jak vyzkoušet aplikaci v simulovaném prostředí, je použít utility[63] společnosti SimplySim, která sama dokáže služby najít a spustit ve svém prostředí. Navíc umožňuje robota ve virtuálním prostředí řídit a aplikaci případně rovnou spustit i v reálném robotovi.

Tuto možnost ale máme, pouze pokud nepoužíváme verzi MRDS Express, protože služby musíme před použitím nejprve zkompilovat, což tato verze neumožňuje. Abychom tedy zachovali co nejvyšší univerzalitu tohoto textu, pouze si tuto možnost ukážeme, ale dále ji používat nebudeme.

Nejprve tedy musíme provést kompilaci naší aplikace do DSS služby. K tomu použijeme volbu menu *Build/Compile as a Service*. Pokud máme diagram sestavený stejně jako na obrázku 33, zřejmě kompilátor ohlásí chybu převodu typu **int** na **double**. Tento problém má na svědomí zadání úhlu jako celého čísla bez desetinné tečky do bloku *Data*. Pravděpodobně se jedná o chybu v MRDS, protože jak typ bloku *Data*, tak očekávaná hodnota v GDD jsou stejného typu, takže někde uvnitř možná dochází ke zbytečnému přetypování. V každém případě pomůže doplnit číslo na *180.0*. Nyní již kompilace proběhne úspěšně a vzniknou knihovny *dll* samotné aplikace i její proxy, které editor rovnou

[63] Nejde přímo o utilitu, ale o běžnou službu navrženou tak, že dokáže vyhledávat v adresáři služeb zadaného DSS uzlu a tyto služby pak spouštět.

umístí do podadresáře *bin* instalace MRDS.[64] Editor se současně zeptá na místo, kam chceme uložit zdrojové kódy nových služeb (v C#), abychom je později měli možnost editovat.

Poznámka 7.3 Můžeme se setkat i se situací, kdy kompilace selže a jako důvod uvede nemožnost přepsat knihovnu *dll*. V takovém případě pomůže restart editoru VPL.

Nyní spustíme službu *NXT-MSRDS*, nejlépe přes zástupce v nabídce start systému Windows, kde jsou již nastaveny potřebné parametry příkazové řádky pro DSS uzel.

Ve spodní části okna aplikace stiskneme tlačítko *Add* a ze seznamu služeb vybereme naši čerstvě zkompilovanou. Ta se poté objeví v seznamu připravených služeb společně s ukázkovými službami dodanými SimplySim v rámci jejich produktu. Nyní můžeme stisknout tlačítko *Run in simulation* a v okně virtuálního prostředí uvidíme robota provádět stejnou akci, jako když jsme v tomto prostředí spouštěli službu přímo z VPL editoru. Navíc můžeme pomocí trackballu robotem pohybovat a současně sledovat stavy jeho senzorů.

Poznámka 7.4 Může se stát, že služba v seznamu nebude. V takovém případě je nutné ručně smazat soubor *contractDirectoryCache.bin* v podadresáři *store\cache* instalace MRDS.

Okno aplikace včetně robota ve virtuálním prostředí a okno DSS uzlu si můžeme prohlédnout na obrázku 35. Červená chybová hlášení upozorňují, že se nepodařilo připojit k reálnému HW robota.

Pokud máme k PC připojeného i reálného robota, můžeme nastavit parametry připojení (měl by stačit pouze sériový port, který emuluje Bluetooth připojení) a stejnou službu můžeme spustit i v reálu. V případě, že se aplikaci podaří s robotem navázat spojení, robot krátce pískne a následně provede otočku.

Poznámka 7.5 Tuto aplikaci můžeme s výhodou použít i bez programování vlastních služeb, například pokud potřebujeme zkalibrovat senzory, ověřit si funkce motorů apod. Hodnoty přečtené ze senzorů pak můžeme použít přímo ve své aplikaci, aniž bychom se zdržovali opakovanými pokusy „naslepo“.

7.6.5 Integrace dashboardu

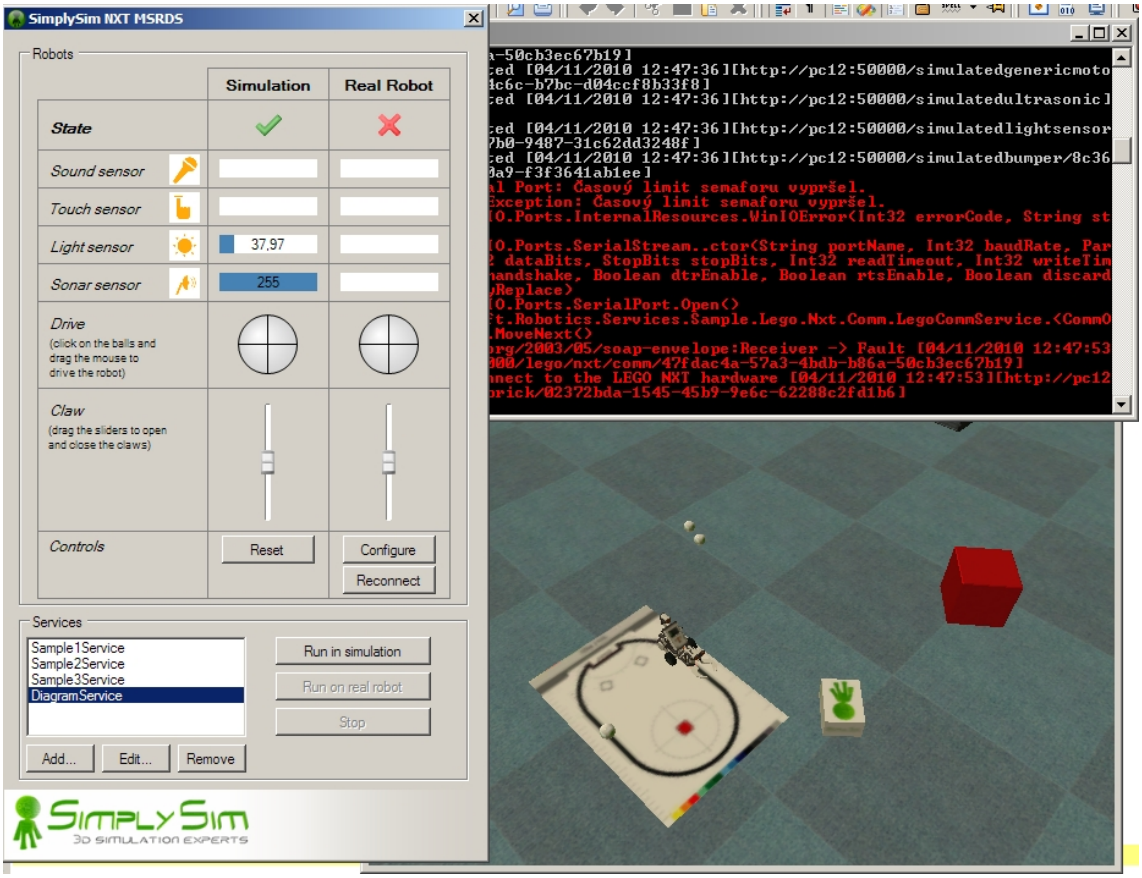
V předchozí části jsme viděli, že může být výhodné robota ovládat ručně i za běhu námi vyvíjené aplikace. Kostra okna aplikace SimplySim je odvozena od tzv. *Simple Dashboardu*, což je opět jedna z předkompilovaných služeb dodaných společně s MRDS. Proto nám nic nebrání tuto službu zařadit i do své aplikace přímo a zpřístupnit si tak podobné funkce.

Podrobnější popis možností a ovládání *Simple Dashboardu* nalezneme v nápovědě MRDS, zde si pouze ukážeme, jak službu začlenit a jak použít.

Začlenění služby je triviální. Prostě blok *Simple Dashboard* přeneseme z palety služeb do našeho diagramu.[65]

[64] Můžeme si všimnout, že nová služba se okamžitě objeví i v paletě služeb v levém okně editoru. Pokud ne, pomůže v menu vybrat volbu *View/Reload Services*.

[65] Protože je tato služba na naší aplikaci zcela nezávislá, vlastně tím jenom editoru řekneme, že ji chceme spouštět společně s aplikací.



Obrázek 35: Okno aplikace SimplySim

Použití není o moc složitější. V okně dashboardu stiskneme tlačítko *Connect* a v okně pod ním se nám zobrazí aktuálně spuštěné služby, ke kterým se můžeme připojit. Pravděpodobně uvidíme pouze jednu, naši aplikaci, takže spojení dvojklikem navážeme. Následně stiskneme tlačítko *Drive*, čímž aktivujeme odesílání povelů přímo naší službě. Pohyb motoru robota pak ovládáme trackballem stejně jako v případě aplikace SimplySim.

Poznámka 7.6 Dashboard využijeme nejčastěji při ladění aplikací, kdy potřebujeme robota nasměrovat do nějaké pozice, vrátit zpět apod.

7.6.6 Úpravy scény

Nyní už známe dvě prostředí, ve kterých můžeme naše aplikace spouštět. Protože tvorba zcela nového prostředí není úplně jednoduchá, zkusíme si vystačit s úpravami těch stávajících.

Například ve výchozím prostředí MRDS vidíme několik objektů, které jsou ale od robota poměrně vzdálené a v dalších úlohách budeme chtít, aby robot nějak interagoval

s okolím. V současné podobě bychom ho tedy museli nejprve nasměrovat k cíli a počkat, až k objektům dojede. Výhodnější tedy bude scénu upravit tak, abychom měli robota k cíli co nejblíže.

V kapitole 7.6.1 jsme prozkoumali manifest, který pro spuštění této simulace používáme. Mimo jiné jsme tam našli odkaz na další XML soubor *LEGO.NXT.Tribot.SimulationEngineState.xml*. Právě v tomto souboru je uložen aktuální stav scény, který se má při spuštění prostředí nastavit. Pokud si jej otevřeme v textovém editoru, najdeme popis jednotlivých objektů scény, včetně jejich polohy, textury apod. Nicméně v takovém množství dat není jednoduché se orientovat. Proto existuje i snadnější možnost.

Spustíme si tedy naše virtuální prostředí (můžeme společně s naší aplikací nebo samostatně přes nabídku start systému Windows, kde jsou pro každé prostředí vytvořeni zástupci) a v menu zvolíme *Mode/Edit*. Nyní můžeme volně editovat všechny objekty ve scéně. Nejsnazším způsobem je kliknout pravým tlačítkem na objekt a z kontextové nabídky zvolit například *MoveXZ*, pokud požadujeme pohyb v rovině podložky (dopředu,dozadu a do stran). Nyní stiskneme klávesu *Ctrl* a vybraný objekt bude zvýrazněn vystínovaným kruhem. Teď ho můžeme myší posunout na požadované místo.

Poznámka 7.7 Před vlastní editací je nanejvýš vhodné se seznámit s ovládáním editoru virtuálního prostředí, zejména se směry pohybu a způsoby jejich změn. Ušetříme si tím spoustu času a marných pokusů.

Pokud jsme s výsledkem spokojeni, můžeme ještě nastavit vhodný pohled kamery a scénu uložit volbou *File/Save scene as*. Zjistíme, že nám editor nepovolí uložit scénu mimo adresářovou strukturu MRDS, proto si v něm vytvoříme vlastní podadresář *Scenes* a do něj scénu necháme uložit.

Poznámka 7.8 Při editaci scény, zejména pokud v ní nejsme ještě zběhlí, je vhodné přepnout režim zpět do aktivní simulace (*Mode/Run*) a zkusit pomocí dashboardu s robotem pohybovat. Vyhneme se tak možným problémům, pokud bychom například robota omylem postavili na nějakou překážku, která by mu bránila v pohybu.

Za předpokladu, že jsme si nejprve připravili scénu a až následně budeme tvořit aplikaci, případně v aplikaci editujeme scénu poprvé, je další krok velmi jednoduchý. Do stejného adresáře (pro snadnou orientaci) *Scenes* zkopírujeme manifest simulace, který bychom použili normálně, tzn. např. *LEGO.NXT.Tribot.Simulation.manifest.xml* (raději jej pro snazší vyhledávání přejmenujeme) a v něm odkaz na *LEGO.NXT.Tribot.SimulationEngineState.xml* nahradíme odkazem na náš XML soubor se scénou. Nyní pouze při importu manifestu pro generické služby naší aplikace zvolíme tento upravený manifest. Spuštěnou aplikaci pak uvidíme již v novém prostředí.

V případě, že již máme upravený manifest v aplikaci naimportovaný a provádíme další změny scény, musíme postupovat trochu jinak. Editor VPL totiž všechny XML soubory, na které narazí, kopíruje do adresáře aplikace. Bohužel ani to není všechno. Při prvním spuštění aplikace si dočasně vytvoří pro novou službu unikátní podadresář instalace MRDS a soubory zkopíruje i do něj. Je proto třeba v adresáři aplikace zkopírované XML soubory (ty, se kterými nyní pracujeme) smazat a odstranit na ně odkazy v konfiguraci projektu. To uděláme v pravém horním okně editoru v sekci *Configurations*. Nyní

v našem upraveném manifestu změníme odkaz na nově uloženou scénu a provedeme opakovaný import tohoto manifestu. V některých případech může být nutné před touto poslední fází editor VPL ukončit a znovu spustit. Výsledkem by každopádně měla být aplikace spouštěná v nové scéně.

7.7 Reálný robot

Nyní, když jsme se seznámili s možnostmi virtuálního testování, přistupme k tvorbě aplikací pro reálného robota. Také zde máme více možností jak postupovat.

Nejjednodušší je opět využít dodané generické služby a k nim použít vhodný manifest, který zajistí komunikaci s robotem. I tyto manifesty jsou součástí balíku MRDS. Například pro spuštění naší první úlohy postačí importovat nový manifest *LEGO.NXT.TriBot.Manifest.xml*. Než tak však učiníme, musíme ještě nastavit správné parametry našeho robota.

Nejdůležitější informací je sériový port, který emuluje systém Windows pro spojení s robotem. Když si prohlédneme výše zmíněný manifest, najdeme v něm odkaz na službu *NXT.Brick*, jejíž konfigurační soubor je *LEGO.NXT.Brick.Config.xml*. Brick (Kostka) tedy musí být spuštěna společně s dalšími službami, aby mohly s robotem komunikovat. Právě v tomto XML souboru najdeme element *SerialPort*, jehož hodnotu bychom měli nastavit.

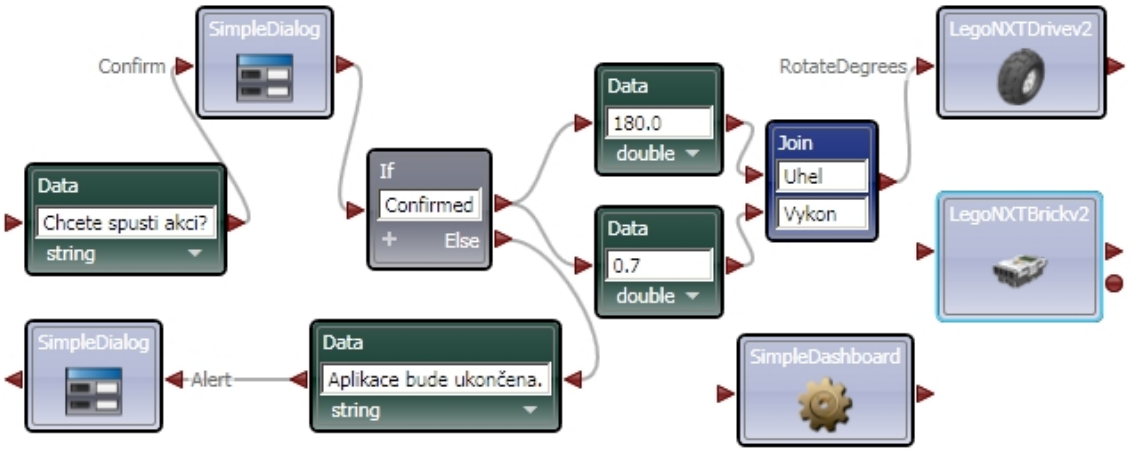
Další změny musíme udělat v konfiguračním souboru pro ovládání diferenciálního pohonu. Tentokrát jde o soubor *LEGO.NXT.TriBot.Drive.Config.xml*. Zde bychom měli správně nastavit velikost (průměr) kol a vzdálenost mezi nimi. Pravě ta je defaultně nulová, což je zdrojem častých problémů při výpočtu úhlů a vzdáleností.

Poznámka 7.9 Je třeba si uvědomit, že zatímco manifesty simulací počítají vždy s týmž modelem robota (Tribot), proto mají všechny parametry již správně nastaveny, zde máme k dispozici univerzální služby pro ovládání jakéhokoli modelu robota. Ačkoli v tomto textu budeme i v reálu používat Tribota, nemusí tomu tak být obecně a vhodné parametry je třeba do konfigurací přenést. Samozřejmě v takovém případě je vhodné vytvořit zcela nové manifesty a vhodně je pojmenovat.

Po provedení těchto změn můžeme manifest importovat a pokusit se aplikaci spustit. Pokud je robot připojen, měl by po startu služby pípnout a provést očekávanou otočku. Současně se v okně prohlížeče objeví stav služby *NXT.Brick*, odkud můžeme online sledovat i stavy jednotlivých spolupracujících služeb. Pokud například zvolíme službu NXT.TriBot.Drive, uvidíme i graficky efektně znázorněný pohyb obou motorů, včetně aktuální rychlosti apod. To je jeden z případů, kdy služba umí odpověď na požadavek *HTTP Get* zabalit do uživatelsky přívětivé podoby.

Další možností je opustit generické služby a použít přímo služby určené pro díly stavebnice LEGO® MINDSTORMS® NXT. Sice přijdeme o možnost simulací a o univerzálnost naší aplikace, ale zase budeme mít možnost využít všechno vybavení robota a i příslušná konfigurace bude pohodlnější.

Poznámka 7.10 Před dalším krokem bychom si měli aplikaci uložit pod jiným jménem, pokud se budeme chtít k předchozím verzím ještě vrátit. Editor VPL totiž při spuštění aplikace automaticky, bez žádosti o potvrzení, přepisuje původní zdrojové soubory.



Obrázek 36: VPL_CelemVzadLEGO – výsledný diagram

Nejprve tedy odstraníme GDD. Následně do diagramu vložíme službu *Lego NXT Brick (v2)* a *Lego NXT Drive (v2)*. Konfiguraci Kostky tentokrát nebudeme provádět přes manifest (ačkoli i to by bylo možné), ale zvolíme *Set initial configuration* a do následně rozbalených políček zadáme potřebné hodnoty (pravděpodobně pouze číslo portu). Stejnou operaci provedeme s druhou službou, kde už ale měněných hodnot bude víc. Nejprve musíme „spárovat“ pohon s Kostkou, v roletě *Partners* tedy najdeme službu Kostky. Dále musíme správně přiřadit oba motory portům na Kostce. V případě Tribota je levé kolo na portu C, pravé pak na portu B. Nakonec stejně jako u manifestu zadáme velikosti a vzdálenost kol.

Zbývá pouze zařadit *Lego NXT Drive (v2)* na správné místo diagramu a nastavit příslušné parametry zprávy stejně jako u GDD. Výsledný diagram vidíme na obr. 36.

Takto připravenou aplikaci nyní můžeme spustit a výsledek by se neměl lišit od výše popsaného řešení s manifesty.

7.8 Úloha 6 – Vpřed 1 m a stůj

Nyní tedy známe několik variant, jak své robotické aplikace odzkoušet ve virtuálním prostředí i v reálu a víme, jak je lze měnit. Můžeme proto pokračovat v dalších praktických úlohách, přičemž ponechme na čtenáři, kterou z možností testování využije.

Poznámka 7.11 Pokud si čtenář vyzkoušel všechny výše zmíněné metody spouštění aplikací, jistě zaznamenal, že jednou importovaný manifest již zůstává v adresáři aplikace a pro jeho opětovné použití jej již není třeba znovu importovat, ale pouze vybrat z rolety. Změna cílové platformy je tak otázkou několika sekund, což oceníme zejména při přechodu mezi reálným robotem a simulací.

7.8.1 Zadání

Po spuštění robot ujede 1 m a zastaví. O provedení úkolu informuje uživatele.[66]

7.8.2 Postup řešení

- **Analýza**
  Úlohu můžeme rozdělit na dvě části. První se stará o pohyb robota a její řešení bude zřejmě podobné jako v předchozí úloze. Druhá má pak za úkol informovat uživatele o dokončení akce.

- **Výběr služeb**
  Také v této úloze požadujeme pouze pohyb robota bez interakce s okolím, proto použijeme tutéž službu GDD a samozřejmě *Simple Dialog*.

- **Sestavení diagramu**
  V předchozím úkolu jsme si vyzkoušeli jednoduchý povel, kterým otočíme robota o určený počet stupňů. Podobnou možnost nabízí GDD také pro ujetí určité vzdálenosti. Tato akce se nazývá *DriveDistance* a jako parametr požaduje ujetou vzdálenost (v metrech) a výkon[67] (stejně jako u otáčky v rozsahu od -1 do 1). Můžeme tedy sestavit stejný diagram jako v předchozím úkolu, pouze změníme použitou akci bloku GDD.

- **Spuštění**
  Pokud jsme provedli změny správně, měl by robot po spuštění aplikace a potvrzení uživatele opravdu ujet jeden metr[68] a zastavit. My však požadujeme ještě informování uživatele o dokončení akce.

[66] Zatím neřešíme detekci překážek, takže u reálného robota je vhodné postavit ho do volného prostoru.

[67] GDD rovněž umožňuje nastavit i rychlost (speed) motoru, která se udává v metrech za sekundu. My ale absloutní hodnotu rychlosti znát nemusíme a rozsah výkonu spíše odpovídá povaze úloh, ve kterých potřebujeme měnit rychlost relativně.

[68] Přesnost je dána mimo jiné správným nastavením průměru kol v manifestu, dále pak skutečnými fyzikálními vlastnostmi robota a povrchu. Pro naše účely stačí přesnost přibližná.

- **Úpravy**
  Logické by se zdálo zapojit na výstup GDD uživatelský dialog se zprávou. Můžeme to tedy vyzkoušet a sledovat, jak bude aplikace reagovat. Pokud používáme aktuální verzi MRDS a robota ze stavebnice LEGO® MINDSTORMS® NXT, případně některou z jeho vizualizací, zjistíme, že potvrzení akce přijde ihned po zahájení pohybu robota.

  To je dáno implementací GDD ve skutečném HW resp. vizualizaci, kdy je doporučeno, aby na výstupní port byla zasílána potvrzení požadavků ihned po přijetí, nikoli po skutečném vykonání akce.

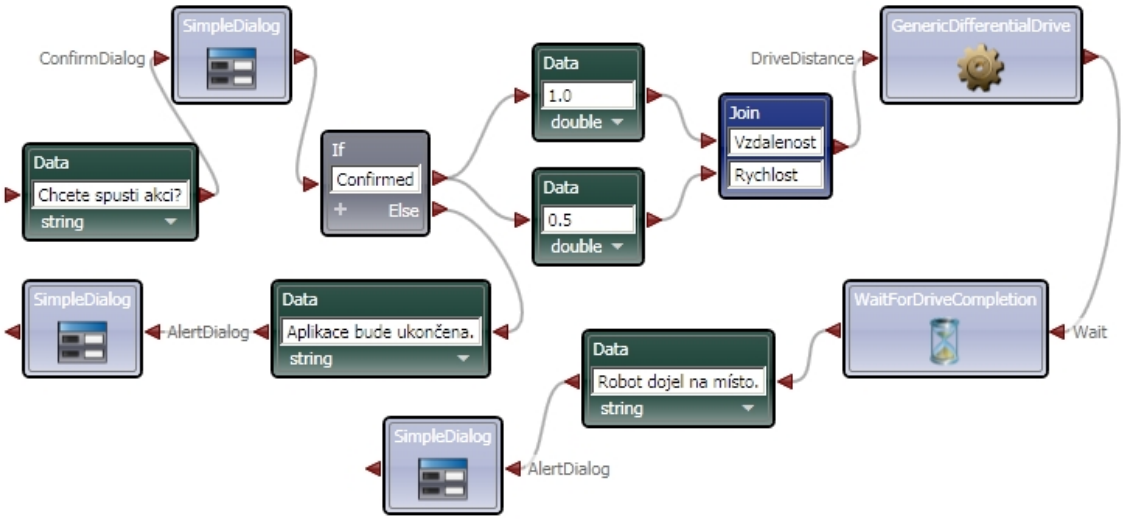
  Poznámka 7.12 Zde došlo ke změně proti prvním verzím MRDS, kdy některé implementace GDD vracely potvrzení teprve po skutečném provedení akce. Na tuto zcela zásadní vlastnost GDD upozorňuje na několika místech dokumentace MRDS (VPL Lab 3 a 4) a podrobně pak v sekci *Robotics Services* v závěru kapitoly *WaitForDriveCompletion*, ovšem ve starších publikacích, které ještě vycházely z MRDS verze 1.5, můžeme najít algoritmy využívající přímé navazování akcí GDD.

  Mohli bychom tedy využít časovač a podle odhadnuté doby provádění příkazu nastavit příslušné zpoždění, to by ale u složitějších aplikací nebylo příliš efektivní.

  Další možností je využít notifikací od GDD, které mimo jiné obsahují zprávy *DriveDistance* a *RotateDegrees*, přičemž po dokončení akce vyšlou zprávu s hodnotou *Completed*. Tento způsob je využit právě v tutoriálu dokumentace MRDS v kapitole VPL Lab 3, proto ho zde nebudeme opakovat.

  Ukážeme si jednodušší způsob, který využívá za tímto účelem tvůrci MRDS speciálně vytvořené aktivity *WaitForDriveCompletion*. Jde o jednoduchou třídu napsanou v C#, která má za úkol naslouchat výše zmíněným notifikacím a po obdržení zprávy *Completed* vyslat zprávu na svůj výstupní port. Mimo jiné nám umožní v dalších úlohách čekat na dokončení akcí i uvnitř uživatelsky definovaných aktivit, kde by použití metody založené na notifikacích nebylo možné (viz poznámka 5.22).

  Výsledný diagram takto upravené aplikace vidíme na obr. 37.



Obrázek 37: VPL_Vpred1m – výsledný diagram

7.9 Úloha 7 – Vpřed 5 s a stůj

7.9.1 Zadání

Po spuštění robot pojede 5 sekund vpřed, zastaví a informuje uživatele.

7.9.2 Postup řešení

- **Analýza**
  Zde můžeme použít opět základní kostru z předchozích úloh, opět jde o pohyb robota s následnou zprávou uživateli. Tentokrát ale o dokončení operace nerozhodne robot sám (GDD), ale vnější vliv, tedy čas. Použití časovače však již známe, proto můžeme zkombinovat již dříve prověřené algoritmy.

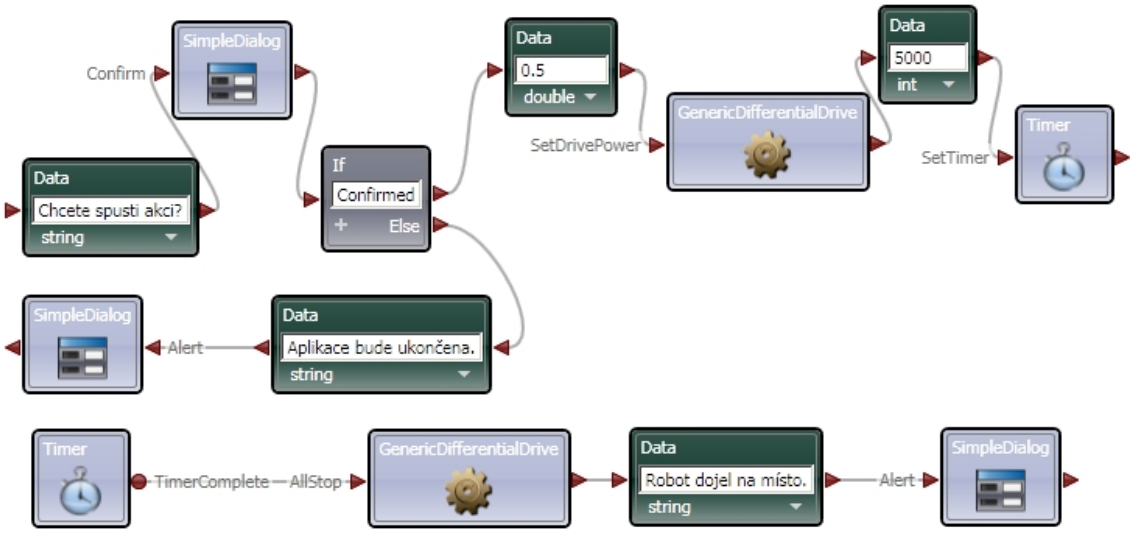
- **Výběr služeb**
  Pro pohyb robota budeme tradičně potřebovat GDD, *Simple Dialog* a tentokrát nově i službu *Timer*.

- **Sestavení diagramu**
  Na rozdíl od předchozích úloh nám bude stačit předat GDD pouze jeden parametr, a to výkon, kterým chceme robota rozjet. K tomu využijeme akci *SetDrivePower*. Robot se okamžitě rozjede.

  Pokud si prohlédneme podrobně všechny akce, které GDD nabízí, najdeme tam i *AllStop*. Právě tu můžeme použít k okamžitému zastavení robota. Stačí nám tedy spojit notifikační výstup časovače se vstupem GDD a zaslat příslušnou zprávu. Na výstupní port pak připojíme informační dialog.

  Poznámka 7.13 Pokud pošleme robotovi pokyn *AllStop*, nejenom že zastaví motory, ale také zakáže jakýkoli další pohyb, dokud se výslovně nepovolí příkazem *EnableDrive*. V některých případech proto může být vhodnější pouze nastavení nulového výkonu motoru, což bude mít stejný efekt, pokud tuto hodnotu pošleme oběma motorům GDD. V jiných situacích (např. při pohybu na základě notifikací) ale může být zablokování dalšího pohybu žádoucí.

- **Spuštění**
  Takto sestavený diagram je znázorněn na obr. 38 a již při prvním spuštění zjistíme, že plní stanovený úkol přesně a není třeba provádět další úpravy.



Obrázek 38: VPL_Vpred5s – výsledný diagram

7.10 Úloha 8 – Patrola

7.10.1 Zadání

Spojením předchozích úkolů dosáhnout toho, aby robot ujel určitou vzdálenost, otočil se, vrátil na výchozí místo, tam se znovu otočil atd.

7.10.2 Postup řešení

- **Analýza**
  Nyní již máme všechny dílčí znalosti k vyřešení tohoto úkolu. Nejprve robota rozjedeme 1 m směrem vpřed, poté provedeme otočku „čelem vzad“ a opět pojedeme 1 m vpřed. Abychom mohli robota zastavit, přidáme jednoduchý informační dialog, který na náš pokyn zašle GDD zprávu *stopAll*.

- **Použité služby**
  V této úloze po robotovi požadujeme pouze pohyb, proto nám bude stačit služba GDD a dále informační *Simple Dialog*.

- **Sestavení diagramu**
  Vlastně nám bude stačit mezi dílčí operace robota vložit jeden blok *WaitForDrive-Completion*, čímž zajistíme plynulé navázání a týmž blokem po dokončení otočky přivedeme datový tok zpět na vstup GDD.

  Část kódu, která bude provádět „nouzové“ zastavení bude zcela nezávislá.

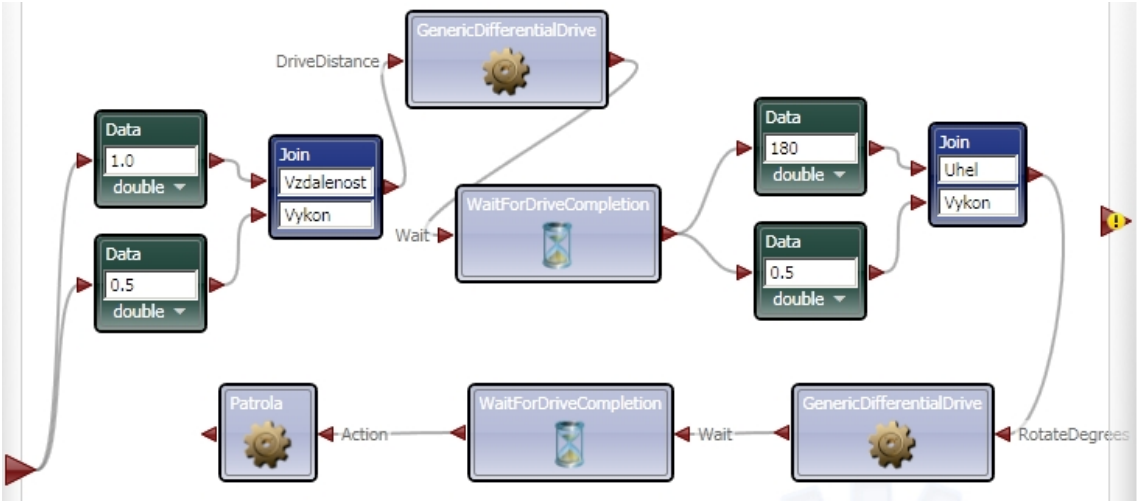
- **Spuštění**
  Aplikace sestavená podle předchozího návrhu je sice funkční, ale velmi nepřehledná. Objevují se v ní smyčky, které však již umíme eliminovat za pomocí rekurze.

- **Úpravy**
  V tomto případě dokonce nemusíme ani testovat žádnou podmínku, vytvoříme tedy (za jiných okolností nebezpečnou) nekonečnou smyčku, kterou přerušíme pokynem zvenčí.

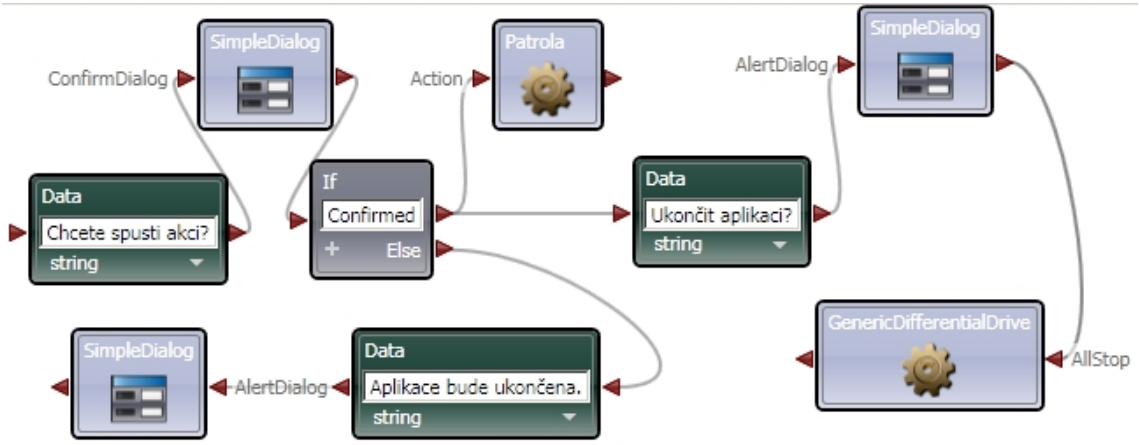
  Abychom mohli rekurzi využít, musíme si vytvořit novou aktivitu a veškerý pohyb robota umístit do ní. To by nám ovšem nemělo činit žádný problém. V hlavním diagramu pak pouze provedeme spuštění operace a případně její zastavení.

  Na obrázcích 39 a 40 vidíme detail nové aktivity resp. celkový pohled na výsledný diagram.

- **Námět na vylepšení**
  Už v předchozím bodě jsme naznačili, že použití nekonečné smyčky není příliš obvyklá metoda a v praxi bychom se jí měli spíše vyhýbat. Proto by bylo vhodnější vytvořit v naší aktivitě další akci, nazvanou třeba *Stop*, která by na pokyn zvnějšku ukončila provádění operace.



Obrázek 39: VPL_Patrola – aktivita *Patrola*



Obrázek 40: VPL_Patrola – výsledný diagram

Poznámka 7.14 Na tomto místě ještě poznamenejme, že záleží na konkrétní implementaci GDD v příslušném HW, jak bude se zprávami nakládat. Ve většině případů poslední zaslaná zpráva (příkaz) ruší všechny předchozí, proto pokud bychom zaslali robotovi pokyn k otočce dříve než po ujetí stanovené vzdálenosti, provede otočku a dál v jízdě nepokračuje. Otočka je poslední pokyn, který ruší předchozí pokyn k jízdě vpřed.

7.11 Úloha 9 – Při nárazu couvej

7.11.1 Zadání

Robot se rozjede proti překážce, po nárazu do překážky informuje uživatele a začne couvat.[69]

7.11.2 Postup řešení

- **Analýza**
  Tentokrát bude pohyb robota ovlivněn dalším vnější příčinou, nárazem na překážku. Budeme tedy muset použít první čidlo, kterým bude tlakový senzor (bumper). K dispozici máme opět jeho generickou verzi, proto i s ním můžeme pracovat ve virtuálním prostředí. Místo zastavení budeme požadovat, aby robot změnil směr jízdy. Toho dosáhneme posláním záporné hodnoty výkonu GDD v okamžiku detekce nárazu.

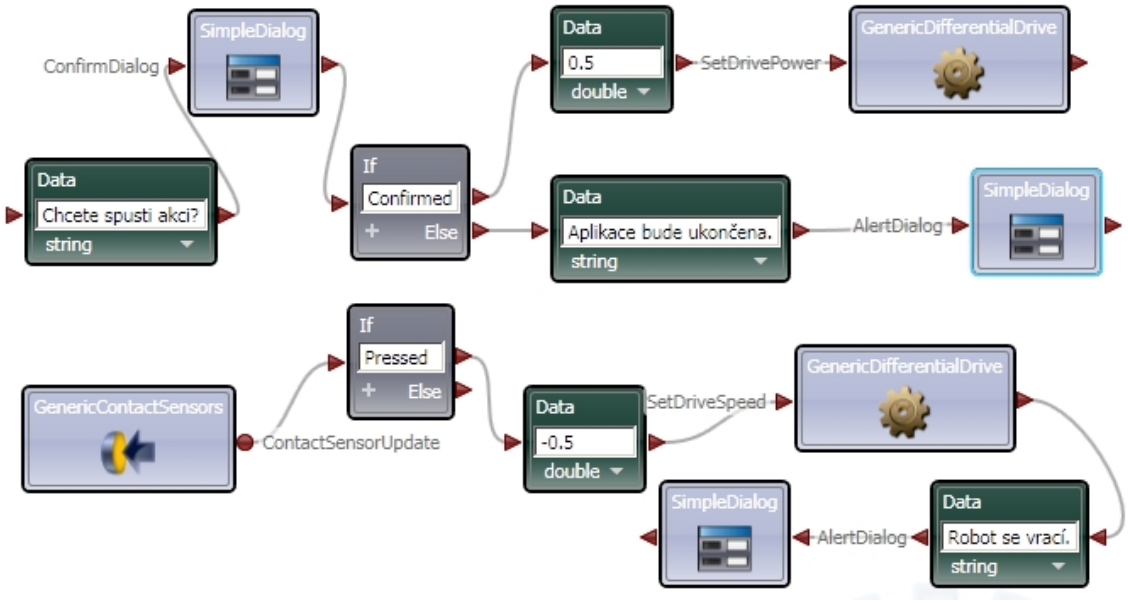
- **Použité služby**
  Opět využijeme GDD, dále *Simple Dialog* a nově *Generic Contact Sensor*, u kterého nesmíme zapomenout zvolit příslušný manifest, který aplikaci zpřístupní implementaci služby pro konkrétní cílové prostředí či HW.

- **Sestavení diagramu**
  Zde můžeme naplno využít výhody paralelního zpracování zpráv. Požadujeme, aby robot jel nějakým směrem a při aktivaci bumperu začal couvat. Stačí nám tedy vyslat pokyn pro jízdu vpřed a zároveň čekat na notifikace od bumperu. Pokud nějaká dorazí, otestujeme, zda se jedná o jeho aktivaci (*Pressed*), následně změníme výkon obou motorů na záporný a současně informujeme uživatele.

- **Spuštění**
  Výsledný diagram vidíme na obr. 41. Pokud aplikaci testujeme na reálném robotovi, pravděpodobně budeme s výsledkem spokojeni. Bohužel v obou virtuálních prostředích narazíme na menší problémy. V tom výchozím od MRDS nestojí v přímém směru před robotem žádná překážka, proto je třeba nejprve (např. dashboardem) robota nasměrovat. V prostředí SimplySim je zase model robota vybaven čelistmi, které částečně kryjí bumper. V některých případech tedy robot narazí pouze čelistmi a k aktivaci bumperu vůbec nedojde. V obou případech je samozřejmě možná také modifikace prostředí dle vlastních požadavků.

  Poznámka 7.15 V této úloze si také můžeme poprvé všimnout chování robota po ukončení aplikace. Zatímco v těch předchozích končil robot v klidu, případně byl zastaven uživatelem, tady byla posledním pokynem změna výkonu. Dokud nepošleme robotovi pokyn jiný, bude pokračovat v jízdě, dokonce i po ukončení aplikace. Bylo by tedy vhodné zařadit další ovládací dialog nebo dashboard, kterým bychom

[69] Na reálném robotovi můžeme náraz simulovat stisknutím bumperu.



Obrázek 41: VPL_PriNarazuZpet – výsledný diagram

robota mohli před ukončením aplikace zastavit. Zde to ovšem z prostorových důvodů dělat nebudeme.

- **Námět na vylepšení**
  Nemůžeme přímo zařadit smyčku, např. v podobě rekurze, protože robot je vybaven bumperem pouze vpředu. Vylepšit algoritmus pro cestování robota mezi překážkami bychom tedy mohli např. couvnutím o několik centimetrů, provedením pokynu „čelem vzad“ a teprve pak rekurzivním opakováním předchozí činnosti. Všechny dílčí znalosti k tomu již máme a neměl by být problém takto aplikaci modifikovat.

7.12 Úloha 10 – Jeď k překážce a zastav 30 cm od ní

7.12.1 Zadání

Robot se rozjede proti překážce a zastaví 30 cm před ní.

7.12.2 Postup řešení

- **Analýza**
  V této úloze by již měl robot působit poněkud „inteligentnějším“ dojmem, než je průzkum terénu narážením do překážek. K tomuto účelu je vybaven ultrazvukovým senzorem (dále sonar), který na základě odražených vln od překážky umožňuje odhadnout vzdálenost k ní. Detailní technické parametry a omezení nejsou předmětem této práce, proto případné zájemce odkážeme např. na [25].

  Samotná realizace bude velmi podobná předchozí úloze. Notifikace sonaru budou robotovi poskytovat informace o aktuální vzdálenosti k překážce. Při poklesu hodnoty pod nastavenou mez se robot zastaví.

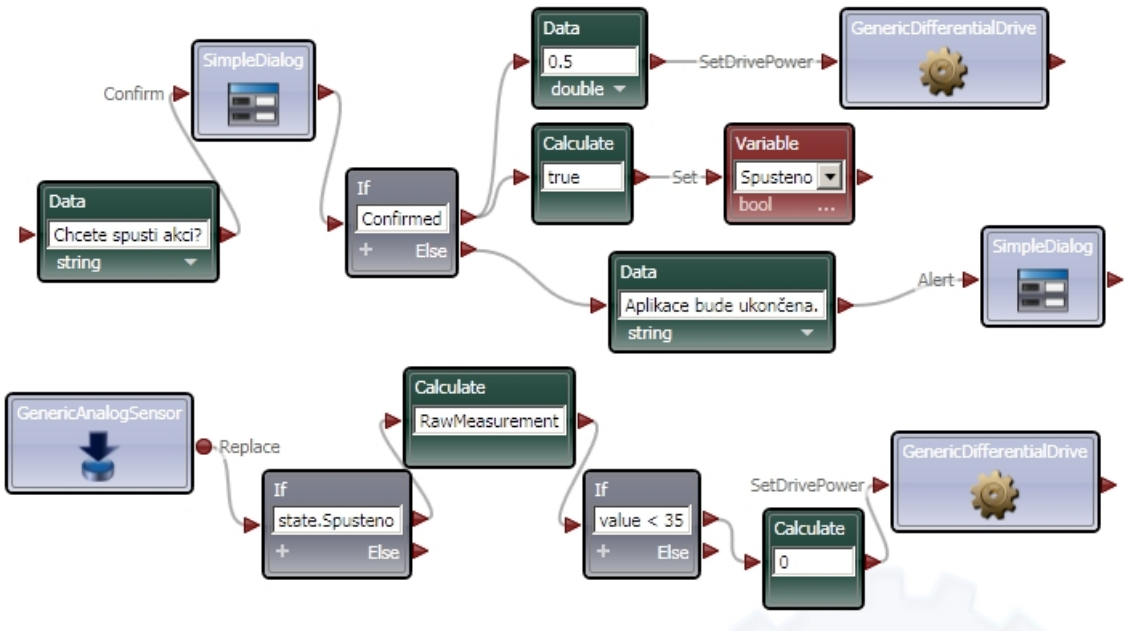
- **Použité služby**
  Mimo GDD a *Simple Dialog* opět použijeme i generický senzor, tentokrát ale *Generic Analog Sensor*. Tento senzor je ovšem implementován pouze ve virtuálním prostředí SimplySim, proto musíme zvolit příslušný manifest nebo použít reálného robota.

- **Sestavení diagramu**
  Diagram si můžeme rozdělit na dvě zcela nezávislé části, podobně jako v minulé úloze. V první části rozjedeme robota a ve druhé budeme vyhodnocovat informace ze sonaru. Rozdílem proti úloze s bumperem je ovšem to, že pokud bychom na údaje ze sonaru vyhodnocovali stejně jako v případě bumperu, robot by se rozjel ještě před potvrzením vstupního dialogu uživatelem. Musíme tedy vložit proměnnou, kterou budeme zprávy filtrovat. Výsledný diagram aplikace je na obrázku 42.

- **Spuštění**
  Pokud aplikaci spustíme ve virtuálním prostředí, patrně nebudeme mít možnost ověřit, jak přesně robot plní zadaný úkol. Naopak v případě reálného robota by bylo velkým překvapením, pokud by se podařilo dosáhnout přesné vzdálenosti 30 cm napoprvé.

- **Úpravy**
  Údaje sonaru jsou závislé jak na povrchu, od kterého se zvukové vlny odráží, tak na přesném nasměrování sonaru samotného. Bude tedy nutné hraniční hodnotu, která zastaví robota, stanovit experimentálně. K tomu nám ideálně poslouží dashboard, jehož integraci do aplikace jsme si ukázali v kapitole 7.6.5.

  Poznámka 7.16 Musíme také vzít v úvahu, kde na těle robota je sonar umístěn a počítat pouze skutečnou vzdálenost robota jako celku, nikoli jen sonaru samotného.



Obrázek 42: VPL_Zastav30cm – výsledný diagram

7.13 Úloha 11 – Jeď k překážce a zpomaluj

7.13.1 Zadání

Robot se bude k překážce přibližovat tím pomaleji, čím je k ní blíž. Ve vzdálenosti 10 cm se zastaví.

7.13.2 Postup řešení

- **Analýza**
  Základ aplikace můžeme převzít z předchozí úlohy. Místo nastavení nulové hodnoty výkonu GDD budeme tentokrát hodnotu plynule snižovat, až k mezní hranici, při které robota zastavíme. Abychom robota zbytečně neusměrňovali v případě, že je překážka příliš daleko, je vhodné zařadit také mez, nad kterou bude výkon konstantní.

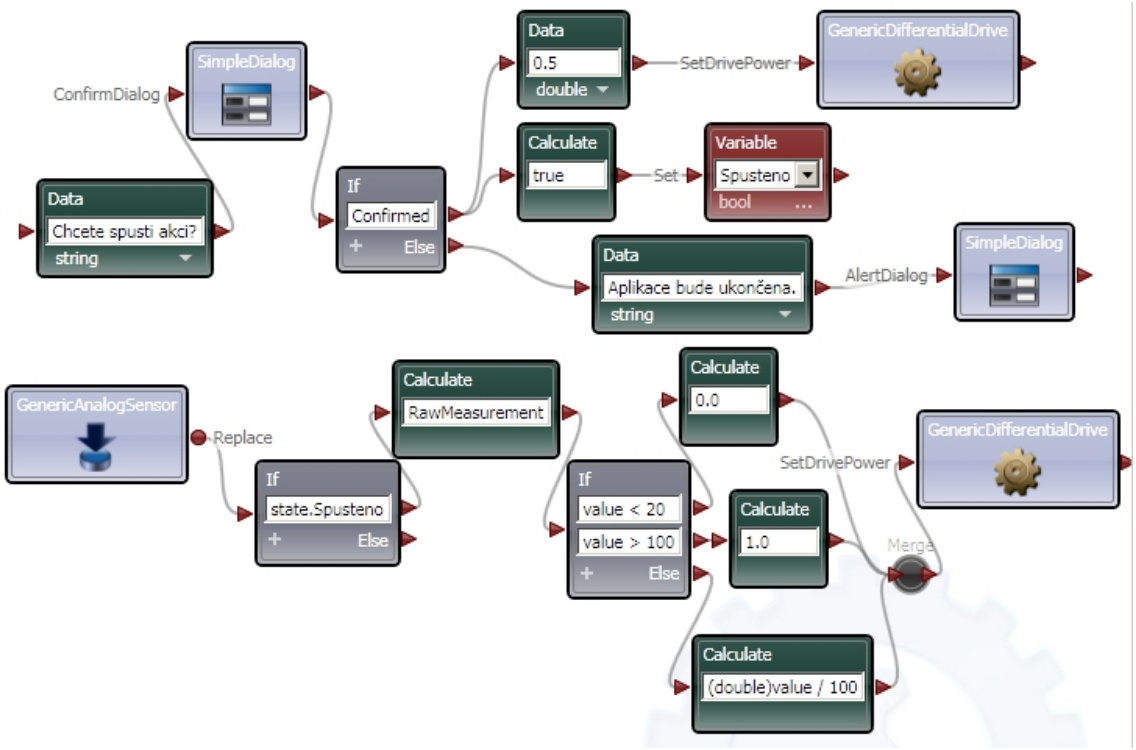
- **Použité služby**
  Použijeme stejné služby jako v předchozí úloze, tedy GDD, *Generic Analog Sensor* a *Simple Dialog*.

- **Sestavení diagramu**
  Znovu rozšíříme předchozí diagram. Místo nastavení nulové hodnoty vložíme podmínku pro obě mezní hodnoty sonaru (ty budeme muset opět určit experimentálně) a pro větev odpovídající „nebezpečné“ vzdálenosti kolem překážky vypočteme novou rychlost robota. Pro horní hranici vzdálenosti lze s výhodou použít hodnotu 100 (pokud nepožadujeme nějakou přesnou), jejíž 1% pak tvoří přesnou hodnotu výkonu. Samozřejmě můžeme použít hodnotu jinou a pak příslušně podíl upravit.

  Výsledný diagram je na obr. 43. Za povšimnutí stojí také přetypování všech hodnot na typ *double*, abychom mohli použít blok *Merge*.

- **Spuštění**
  Stejně jako v předchozí úloze, i zde budeme muset krátkým experimentováním zjistit správné hraniční hodnoty sonaru. Zajímavého efektu také dosáhneme, pokud budeme před robota do vzdálenosti mezi oběma hraničními vkládat nějakou pohyblivou překážku (třeba vlastní ruku). Pokud totiž překážku vzdálíme nebo úplně odstraníme, robot opět zrychlí.

- **Námět na vylepšení**
  Zatím máme nastavenu minimální hodnotu výkonu motoru na 0 (údaj ze sonaru nemůže být záporný). Pokud bychom ale pod nějakou úrovní násobili hodnotu výkonu zápornou konstantou, mohli bychom vytvořit robota, který by „uhýbal“ i pohybujícím se překážkám. Při přiblížení pod kritickou mez by prostě začal couvat. Pozorný čtenář navíc přijde na to, že nestačí hodnotu sonaru převést na zápornou, ale navíc je třeba na její velikost reagovat podle nepřímé úměry, jinak by robot vetřelci těžko unikl. Nejsnáze tedy výsledek vypočteme jako převrácenou hodnotu násobenou nějakou vhodnou zápornou konstantou.



Obrázek 43: VPL_PredPrekazkouZpomaluj – výsledný diagram

7.14 Úloha 12 – Dojeď na čáru

7.14.1 Zadání

Robot se rozjede směrem vpřed a bude sledovat povrch podložky. V okamžiku, kdy dojede na tmavou čáru, se zastaví.[70]

7.14.2 Postup řešení

- **Analýza**
  Tentokrát si vyzkoušíme nový senzor, kterým je robot schopen reagovat na změnu osvětlení. Ve většině aplikací se však využívá pro detekci odraženého světla od povrchu podložky nebo překážky (proto je i vybaven přisvícením), což bude i náš případ. Robot rozpozná čáru podle náhlého poklesu množství odraženého světla. Pro správnou funkci je tedy potřeba, aby hranice mezi čarou a okolím byla dostatečně kontrastní.[71]

- **Použité služby**
  Také pro tento senzor lze v aplikaci VPL použít *Generic Analog Sensor* a stejně jako v případě sonaru i tentokrát se musíme spokojit s testováním v prostředí SimplySim nebo na reálném robotovi. Dále použijeme standardně GDD a *Simple Dialog*.

- **Sestavení diagramu**
  V zásadě lze opět použít již několikrát ověřenou kostru aplikace z předchozích úloh. Pokud správně (opět experimentálně) stanovíme mezní hodnotu senzoru, která rozhodne o zastavení robota, můžeme použít diagram z kapitoly 7.12 (Zastavení 30 cm před překážkou) téměř beze změny.

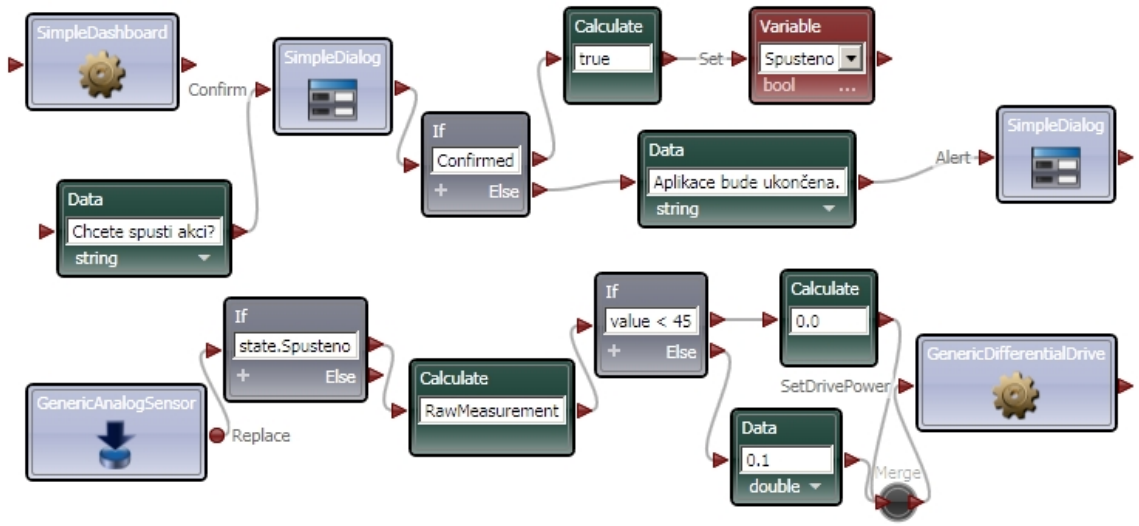
  Přesto si malou změnu vyzkoušíme. Místo úvodního nastavení počáteční rychlosti robota budeme rychlost nastavovat až podle aktuální hodnoty senzoru. Tím si můžeme vyzkoušet, jak bude robot reagovat, pokud například zvolíme rychlost příliš vysokou. Podle podkladu, který použijeme, se může stát, že setrvačností robot i přes vyhodnocení tmavé podložky bude pokračovat v jízdě. Tyto zkušenosti pak můžeme zužitkovat v dalších úlohách.

- **Spuštění**
  Pokud aplikaci testujeme v prostředí SimplySim, asi budeme po prvním spuštění zklamáni, ale robot se ani nepohne. Bohužel pro nás již na černé čáře stojí. Částečně je to i vinou vypuštění počáteční rychlosti robota, ale vzhledem k jeho poloze na čáře by ani v původní verzi daleko nedojel.

- **Úprava**
  Situaci můžeme řešit jako obvykle několika způsoby, z nichž nejelegantnější je asi převrácení podmínky zadání, tzn. zastavení při dojetí na čáru ne černou, ale bílou.

[70] Pravděpodobně bude třeba v každém prostředí zvlášť nastavit rozhodovací hodnoty detektoru světla.

[71] Tato podmínka nemusí být nutně splněna, ale v závislosti na okolním osvětlení (tedy pro senzor šumu) může robot zastavit pokaždé na jiném místě.



Obrázek 44: VPL_DojedNaCaru – výsledný diagram

To uděláme jednoduše obrácením podmínky v bloku *If*. Abychom však dostáli podmínkám zadání kompletně, můžeme do diagramu integrovat dashboard a před potvrzením spouštěcího dialogu jím robota přesunout do volného prostoru.

Takto upravený diagram, včetně vloženého bloku *Simple Dashboard* pro počáteční nastavení polohy robota, vidíme na obrázku 44.

7.15 Úloha 13 a – Sleduj čáru 1

7.15.1 Zadání

Robot stojí na černé čáře představující dráhu a rozjede se vpřed. Přitom se snaží z dráhy nesjet. Předpokládáme, že je přechod mezi drahou a okolím dostatečně kontrastní.

7.15.2 Postup řešení

- **Analýza**
  Pokud se zamyslíme nad tím, jak nejlépe udržet robota na trati, asi nám jako nejvhodnější přijde vybavit ho dvěma senzory po obou stranách trati, přičemž se robot bude snažit, aby pod oběma udržel světlý podklad. Bohužel základní stavebnice LEGO® MINDSTORMS® NXT, a tedy i Tribot, obsahuje pouze jeden světelný senzor, takže musíme zvolit odlišnou strategii.

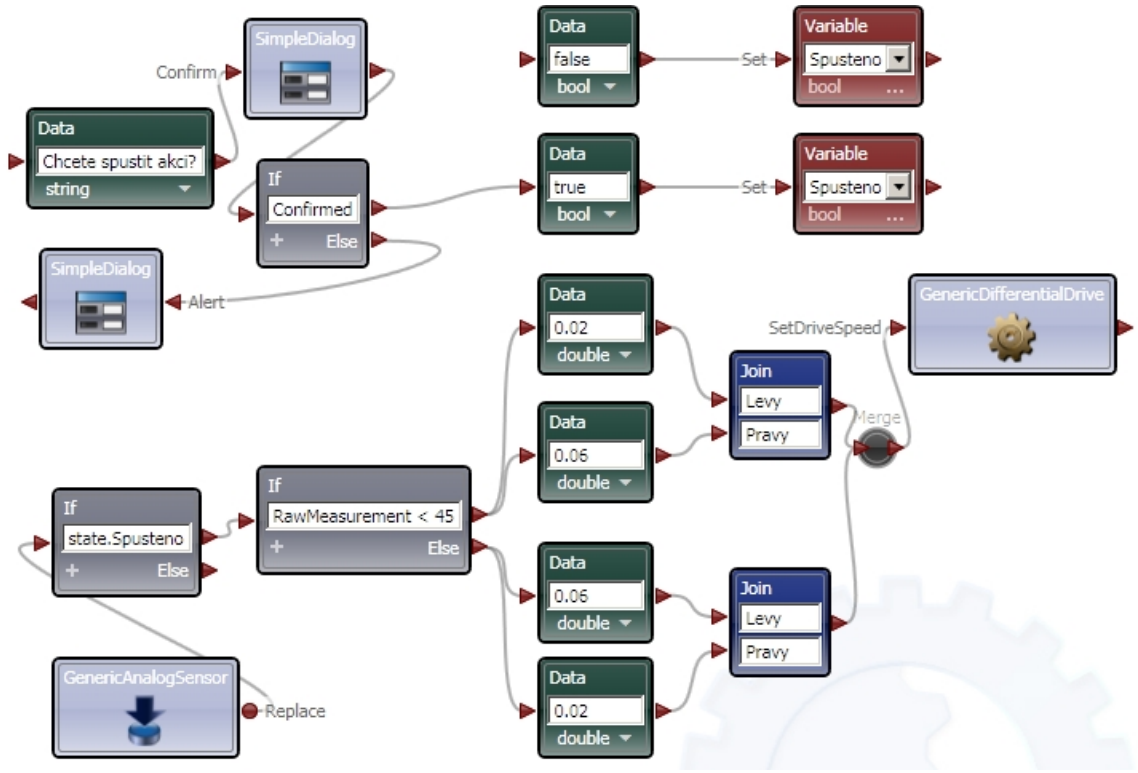
  Budeme muset nechat robota jet po hraně černé dráhy, abychom si byli jisti, že robot vždy ví, na kterou stranu má zatočit. Řekněme, že si zvolíme hranu levou. Potom v případě, že se intenzita odraženého světla od podložky zvýší, robot dráhu opouští a musí zatočit vpravo. Naopak pokud se intenzita sníží, nachází se robot již uvnitř dráhy a musí zatočit vlevo. Za předpokladu, že na trati nebudou žádné křižovatky, tak máme zajištěno, že robot bude dráhu sledovat.

- **Výběr služeb**
  Opět použijeme GDD, *Generic Analog Sensor* pro optický senzor a *Simple Dialog* pro komunikaci s uživatelem.

- **Sestavení diagramu**
  Robot by měl reagovat na hodnoty přečtené ze světelného senzoru. Ty získáme z notifikačního portu senzoru a přivedeme je na vstup bloku *If*, kde musíme rozhodnout, kam má robot zatočit. Stanovíme si tedy zkusmo nějakou hraniční hodnotu a vložíme dvě podmínky. Nejprve pro hodnotu větší nebo rovno pak pro menší než naše zvolená hranice. Asi nás hned napadne, že podmínku můžeme zredukovat a využít výchozí větev *else* bloku *If*.

  Dále musíme zajistit, aby robot provedl potřebnou otáčku. Mohli bychom použít příkaz pro rotaci o nějaký úhel, ale následně bychom museli ještě robota rozjet vpřed. Tím by se stal pohyb trhaným, proto raději budeme zatáčení korigovat změnou rychlosti obou motorů. Je přitom důležité, abychom zachovali nějakou dopřednou rychlost, tzn., že každý z motorů musí mít alespoň minimální kladnou hodnotu výkonu. Konkrétní hodnoty budeme muset stanovit experimentálně až podle výsledků zkušebních jízd.

  Stejně jako v předešlých úlohách přivedeme obě hodnoty výkonu na vstup GDD přes bloky *Data* a *Join*. Počáteční spouštěcí fázi použijeme stejnou jako v předchozích příkladech. Výsledný diagram této úlohy je na obr. 45.



Obrázek 45: VPL_SledujCaru1 – výsledný diagram

Obrázek 46: VPL_SledujCaru1 – robot se zapnutým senzorem v prostředí SimplySim

- **Spuštění**
  Nyní můžeme aplikaci spustit. Protože používáme světelný senzor, který není v základním prostředí simulace MRDS k dispozici, máme na výběr pouze prostředí SimplySim nebo reálného robota. Importujeme tedy příslušný manifest a aplikaci spustíme.

  Protože náš algoritmus je založen na sledování hrany dráhy, musíme na ni robota ještě před spuštěním akce postavit. V opačném případě se nejspíš vydá naslepo do neznáma. Opět můžeme s výhodou použít dashboard.

- **Úpravy**
  Pravděpodobně zjistíme, že hodnoty budeme muset pro každé prostředí a světelné podmínky velmi zásadně měnit. Stejně tak si asi všimneme, že pokud zvolíme příliš vysokou dopřednou rychlost, robot má příliš velký poloměr zatáčení a v ohybech trati se nedokáže vrátit. Naopak pokud nastavíme korekce vzhledem k dopředné rychlosti příliš velké, robot může dráhu přejet a ocitnout se na její opačné hraně, odkud se již neumí vrátit.

- **Námět na vylepšení**
  Možná vylepšení algoritmu vycházejí z hodnocení v předchozím bodě a postupně budou předmětem následujících variant téže úlohy.

7.16 Úloha 13 b,c,d – Sleduj čáru 2,3,4

7.16.1 Zadání

Robot stojí na černé čáře představující dráhu a rozjede se vpřed. Přitom se snaží z dráhy nesjet.

7.16.2 Postup řešení

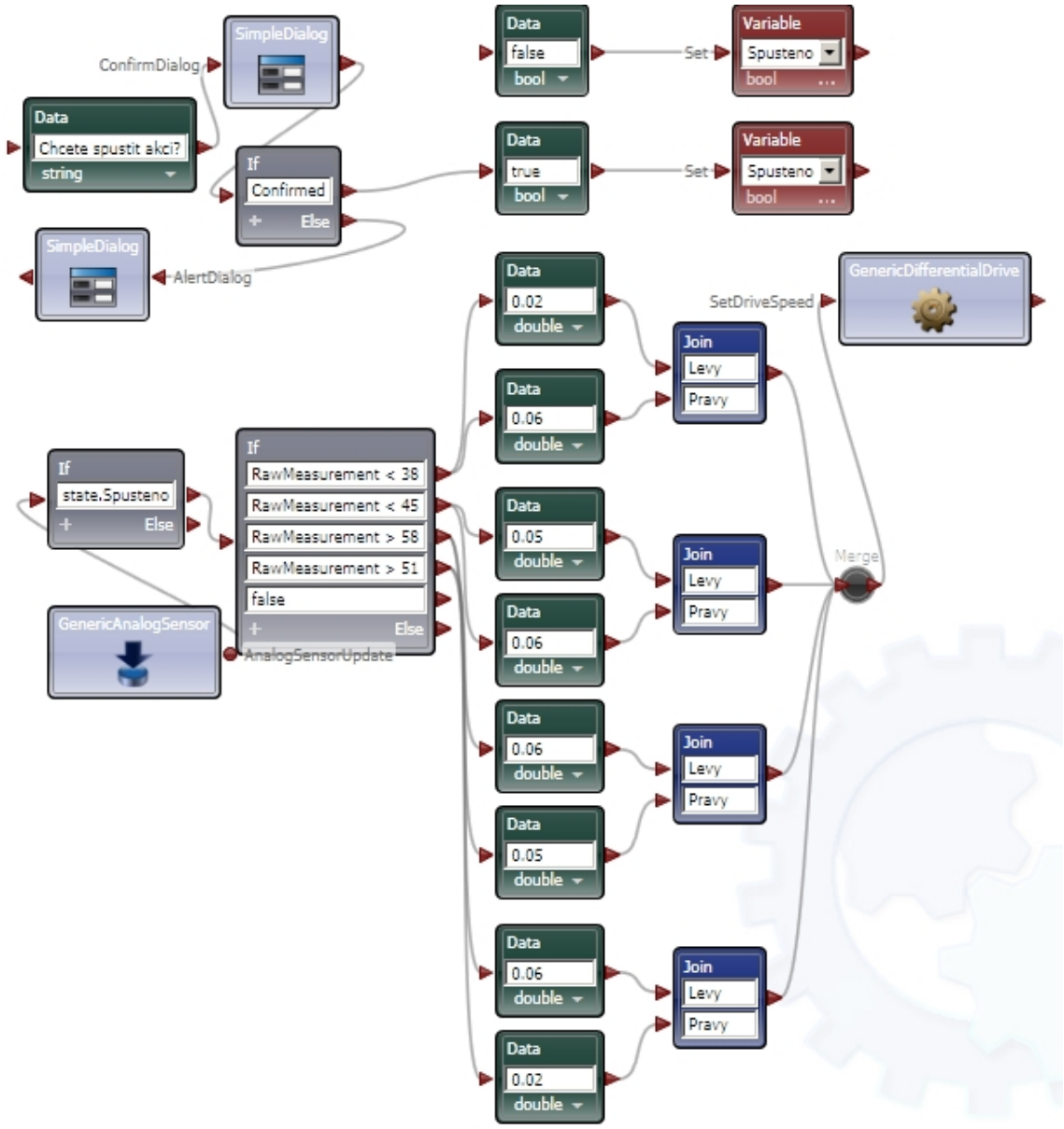
- **Analýza**
  Při hodnocení minulé úlohy jsme se zabývali poměrem korekce dráhy a celkové dopředné rychlosti robota. Když si aplikaci spustíme několikrát, zjistíme, že robot dělá zejména na rovině příliš velké a zbytečné korekce, což se projevuje příliš *klikatou* jízdou. To je způsobeno tím, že dráhu koriguje bez ohledu na velikost odchylky od hraniční hodnoty vždy stejně. Proto v podstatě každý údaj senzoru vyvolá stejně velkou zatáčku.

- **Úprava 1**
  Můžeme tedy například nechat kolem hraniční hodnoty nějakou toleranci, která zachová původní směr robota. Nesmí však být příliš velká, aby se další korekcí robot dokázal vrátit zhruba na hranu dráhy. Takto upravená úloha je na CD nazvaná *VPL_SledujCaru2*.

- **Úprava 2**
  Další možností je škálu údajů senzoru rozdělit na několik částí a v každé stanovit korekci jinak velkou. Čím blíže bude robot hraně dráhy, tím menší korekce budeme provádět. Pochopitelně čím bude rozlišení jemnější, tím přesnějšího vedení robota dosáhneme. Kombinace obou posledních strategií je použita v aplikaci uložené na CD pod názvem *VPL_SledujCaru3*. Její výsledný diagram je na obr. 47.

  Poznámka 7.17 Za zmínku zde stojí použití bloku *If*, kdy jsou podmínky sestaveny tak, aby odpovídaly vyhodnocování směrem odshora dolů. Diagram bychom mohli zjednodušit vynecháním opakujících se bloků *Join*, ale takto je struktura aplikace názornější.

- **Úprava 3**
  Případně můžeme také vyzkoušet strategii, která nutí robota neustále opouštět dráhu a korekcemi podle změny osvětlení se na ni naopak vrací. Princip spočívá v tom, že nemusíme vůbec reagovat na údaje senzoru, které jsou pod střední rozhodovací úrovní. Robot totiž neustále mírně zatáčí do směru mimo dráhu, proto ho tímto směrem posílat nemusíme. Pokud tento algoritmus vyzkoušíme, zjistíme, že zejména na rovinách je velmi stabilní. Robot jede mírně šikmo z dráhy ven, následně je větší korekcí vrácen zpět a opět mírně vybočuje. Problém ovšem může nastat v ostřejších zatáčkách, kdy mohou být korekce nedostatečné. Opět je tedy třeba správnou hodnotu nastavit experimentálně. Takto řešenou úlohu najdeme na CD pod názvem *VPL_SledujCaru4*.



Obrázek 47: VPL_SledujCaru3 – výsledný diagram

7.17 Úloha 13 e – Sleduj čáru s kalibrací

7.17.1 Zadání

Robot si nejprve zkalibruje detektor podle hodnot ve svém okolí a pak se vydá po čáře.

7.17.2 Postup řešení

- **Analýza**
  Protože tato úloha již spojuje několik dříve ověřených postupů, máme šanci odhadnout přibližný rozsah výsledného diagramu. Můžeme sice plochu hlavního diagramu libovolně rozšiřovat, ale v zájmu přehlednosti a oddělení jednotlivých funkčních celků budeme pro některé části diagramu definovat vlastní aktivity.

  Aby robot dokázal rozlišit, zda se nachází na dráze, nebo mimo ni, musí mít nastaveny nějaké rozhodovací hodnoty, jejichž překročení mu signalizuje změnu podkladu. Už v úloze 7.14 jsme zjistili, že je třeba tyto hodnoty nejprve experimentálně zjistit a teprve pak je zadat do programu. Stejně jsme postupovali v předchozích variantách této úlohy.

  V této úloze se pokusíme, aby si robot meze zjistil sám. Zvolíme poměrně jednoduchý postup, který předpokládá, že v určitém výseku kruhového okolí robota se nachází jak nejsvětlejší, tak i nejtmavší bod, se kterým se na trati může setkat.[72] Robot tedy provede rotaci doleva a doprava v rozsahu určeného výseku (zde budeme předpokládat třeba 45 stupňů v obou směrech) a bude zjišťovat hodnoty světelného senzoru. Zajímat nás přitom bude nejsvětlejší a nejtmavší místo. Z nich následně vypočteme střední hodnotu, která by měla odpovídat rozhraní mezi drahou a jejím okolím.[73]

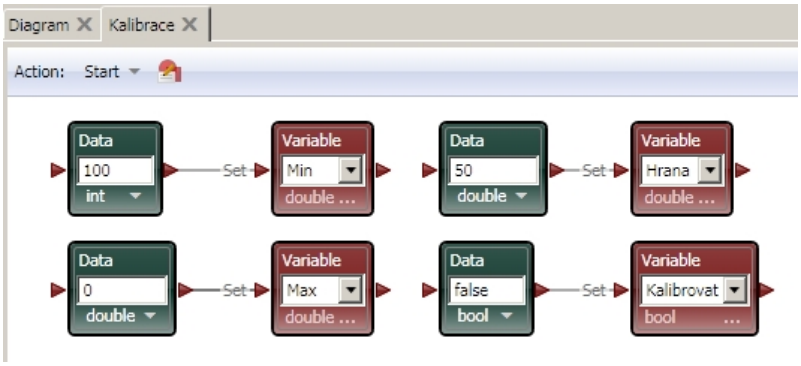
  Následně pak musíme robota nasměrovat do takové pozice, kdy se bude právě na tomto rozhraní, tedy hraně dráhy, nacházet, aby ji mohl začít sledovat. Provedeme tedy zpětnou rotaci a budeme znovu zjišťovat hodnoty ze světelného senzoru. V okamžiku, kdy se přiblíží prahové úrovni, robota zastavíme.

  Poznámka 7.18 Zde budeme muset také trochu experimentovat s rychlostí pohybu a tolerancí prahové hodnoty, protože robot má určitou setrvačnost a také nemůžeme testovat přesně stejnou hodnotu intenzity, jaká nám vyšla výpočtem, protože tu robot nikdy naměřit nemusí (senzor měří a odesílá údaje v nějakých periodách a změny osvětlení tak nedetekuje spojitě).

  Následně spustíme vlastní algoritmus sledování dráhy, který bude velmi jednoduše reagovat na změny ze světelného senzoru. Pokud intenzita poklesne, bude se vracet na dráhu a opačně. Abychom zajistili co nejplynulejší chod robota, nebudeme dělat korekce skokově jako v minulých úlohách, ale plynule dle odchylky od střední

[72] To samozřejmě nemusí být vždy splněno a i u reálného robota zjistíme, že pokud bude dráha v různých místech různě osvětlena, bude robot reagovat nesprávně.

[73] Opět v praxi zjistíme, že to tak úplně neplatí, ale pro tento cvičný úkol nám dosažená přesnost bude vyhovovat.



Obrázek 48: VPL_SledujCaruSKalibraci – akce *Start* aktivity *Kalibrace*

hodnoty osvětlení (což je vlastně do extrému přivedené rozdělení hodnot senzoru na dílčí intervaly). Tedy čím se bude robot více vzdalovat od hrany dráhy, tím ostřejší manévr provede.

Tato úloha je komplikovanější než předešlé, proto si i popis jednotlivých kroků jejího řešení rozdělíme na samostatné sekce odpovídající jednotlivým dílčím aktivitám. V závěrečné fázi pak budeme řešit jejich zapojení do společného diagramu.

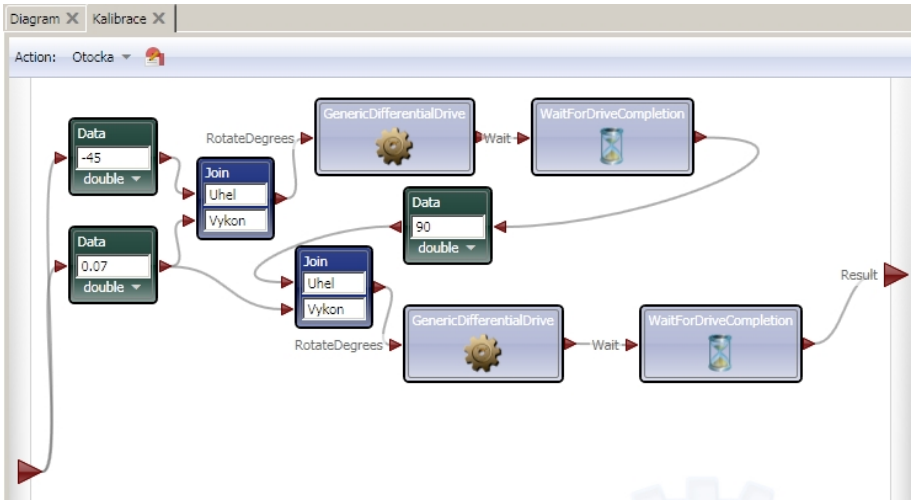
- **Použité služby**
  Stejně jako v předchozích variantách této úlohy použijeme GDD, *Generic Analog Sensor* pro optický senzor a *Simple Dialog*.

- **Kalibrace**
  V této části se zaměříme pouze na první část úkolu, kterým je zkalibrování hodnot optického senzoru. Budeme postupovat podobně jako při návrzích samostatných úloh, proto i členění bude podobné.

  – **Sestavení diagramu**
    Do diagramu si tedy vložíme blok *Activity*. Rovnou si ho přejmenujeme na *Kalibrace*, abychom se později v podobných názvech orientovali. Nyní blok dvojklikem otevřeme v samostatném okně. Novou aktivitu pro využití rekurze jsme již vytvářeli v kapitole 5.11.2. Zde poprvé využijeme i akci *Start*. Jde o výchozí akci, která se provede v okamžiku spuštění aktivity. My zde provedeme inicializaci proměnných.

    Poznámka 7.19 Znovu připomeňme, že proměnné jsou přístupné pouze v rámci dané aktivity nebo dialogu. Nemůžeme tedy provést inicializaci vně aktivity.

    Z rolety *Action* tedy zvolíme *Start* a objeví se nám nová prázdná plocha. Už v analýze problému jsme si řekli, že budeme potřebovat minimální, maximální a střední hodnotu. Všechny si tedy zavedeme (hodnoty senzoru jsou typu **double**, stejného typu tedy budou i naše proměnné) a inicializujeme je na takové hodnoty, které nám zaručí, že je průběh kalibrace změní.



Obrázek 49: VPL_SledujCaruSKalibraci – akce *Otocka* aktivity *Kalibrace*

Současně si připravíme booleovskou proměnnou *Kalibrace*, kterou použijeme na povolování resp. zakazování kalibrace. Také tu inicializujeme na hodnotu *false* (přestože víme, že to není nutné). Tento blok programu vidíme na obr. 48.

Poznámka 7.20 Zdrojem datového toku pro kalibraci budou notifikace od senzoru, které sice můžeme zaregistrovat, ale už ne zrušit. Proto musíme nějakým způsobem filtrovat, zda se zprávy mají dál zpracovávat.

Nyní si vytvoříme vlastní tělo aktivity. Nejprve budeme potřebovat zajistit pohyb robota v požadovaném úhlu kolem svého stanoviště. Tento úhel jsme si stanovili na 45°. Můžeme tedy stejně jako v předchozích úlohách nechat robota udělat otočku o daný úhel v jednom směru. Protože budeme chtít kalibraci provést dostatečně přesně, neměla by být rychlost otáčení příliš vysoká.

Následně provedeme obrat o dvojnásobek úhlu na opačnou stranu. Protože pohyby na sebe musí navazovat, použijeme opět službu *WaitForDriveCompletion*, kterou vložíme mezi obě otočky. Abychom úspěšné dokončení manévru potvrdili teprve po skutečném dokončení pohybu, vřadíme další blok *WaitForDriveCompletion* a teprve jeho výstup spojíme s výstupním portem této akce. Akci v tomto stádiu vidíme na obr. 49.

– **Spuštění**
Můžeme si vyzkoušet, jak bude robot reagovat. Stačí do hlavního diagramu vložit naši známou úvodní část s potvrzujícím dialogem a následně připojit vstupní port nové aktivity *Konfigurace*. Na výběr máme pouze jednu akci, takže navázání proběhne snadno. Nyní aplikaci spustíme, a pokud jsme diagramy sestavili správně, měl by robot vykonat očekávané dvě otočky a skončit pootočený o 45° vzhledem k původní pozici.

Poznámka 7.21 Abychom si usnadnili konfiguraci aplikace, je vhodné všechny služby, které vyžadují import manifestů, vložit mimo aktivity, které je využívají, i do hlavního diagramu (kopie téže instance, nikoli nové) a import provést odtud.

– **Úprava 1 – vyhodnocování senzoru**
Nyní potřebujeme zajistit vyhodnocování přijatých hodnot ze světelného senzoru. Můžeme tedy zkusit do právě vytvářené aktivity vložit blok *Generic Analog Sensor*. Zjistíme ale, že na vložené ikoně chybí notifikační port. To je jedno z omezení, o kterém už padla zmínka v poznámce 5.22 a které souvisí s exkluzivitou paralelně zpracovávaných operací. My ovšem informace od senzoru potřebujeme. Máme tedy dvě možnosti, buď si napsat vlastní DSS službu, u kterých omezení vzhledem k notifikacím neplatí (v nich řeší prioritu procesů programátor a za případné kolize je zodpovědný on) nebo přivést zprávy „zvenku". Snazší cesta bude určitě ta druhá jmenovaná, proto si *Generic Analog Sensor* umístíme do hlavního diagramu.

Pokud se nyní pokusíme připojit notifikační výstup senzoru na vstup aktivity *Kalibrace* (samozřejmě si vytvoříme její novou kopii, ta původní má vstupní port již obsazen), zjistíme, že se nám automaticky zvolil typ akce *Otočka*. My ovšem nechceme, aby se po každém novém údaji senzoru robot znovu otáčel. Proto musíme v naší aktivitě vytvořit novou akci, která bude údaje zpracovávat.

Vedle rolety *Action* tedy klepneme na tlačítko a vyvoláme dialog, ve kterém novou akci přidáme. Pro jednoduchost si ji nazvěme *Kalibruj*. Vidíme, že opět máme prázdnou pracovní plochu. Z toho vyplývá, že bloky v různých akcích spolu nemohou komunikovat přímo, pouze prostřednictvím proměnných nebo rekurze.[74]

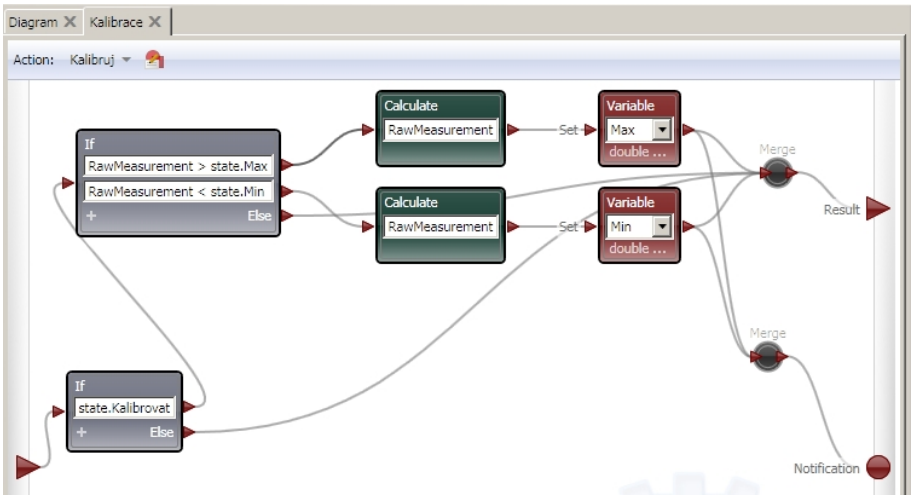
Protože tentokrát budeme potřebovat předávat i hodnotu ze senzoru, rovnou si přidáme potřebný vstupní parametr. Můžeme si ho nazvat stejně jako skutečný parametr senzoru, tedy *RawMeasurement*. Nyní můžeme okno zavřít a vrátit se do hlavního dialogu. Teď už můžeme výstup senzoru napojit na vstup aktivity i s volbou nové akce. Rovnou tedy i přiřadíme správnou předávanou hodnotu.

Zbývá vytvořit vlastní zpracování dat. Jeho úkolem je pouze zaznamenat hraniční hodnoty, stačí tedy testovat vstupní hodnotu proti oběma uloženým a v případě většího extrému hodnotu nahradit.

Ještě jsme si řekli, že budeme testovat, zda máme povoleno kalibraci provádět. Proto do vstupu vložíme příslušnou podmínku. Pro korektní předávání informací o dokončení zpracování všechny volné výstupy přivedeme na výstupní port akce.

Pro účely ladění si rovnou zřiďme i notifikační porty, které nás budou informovat o průběhu kalibrace. Vytvoříme si tedy novou notifikaci (ve stejném okně jako jsme vytvářeli nové akce) a přidáme dva parametry pro minimální a maximální hodnotu. Protože nás bude zajímat pouze změna, přivedeme na

[74] Jak jsme viděli v kapitole 5.11.2.



Obrázek 50: VPL_SledujCaruSKalibraci – akce *Kalibruj* aktivity *Kalibrace*

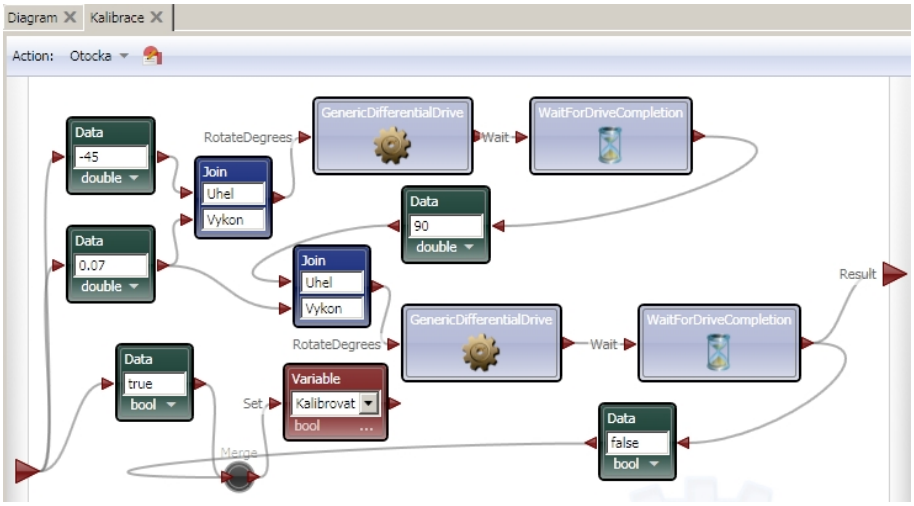
tento notifikační výstup data pouze od obou měněných proměnných. Akci vidíme na obr. 50.

– **Úprava 2 – ladicí informace**
Protože se bude jednat o poměrně složitou úlohu z hlediska koordinace aktivit, už si při ladění nevystačíme se *Simple Dialog*. I pro tyto účely existuje v MRDS přijatelné řešení. Do našeho diagramu můžeme vložit blok *Flexible Dialog*. Jde o velmi zjednodušené okno aplikace, jaké známe z prostředí pro návrh grafických uživatelských rozhraní (GUI). Toto okno můžeme buď konfigurovat za běhu aplikace programově, nebo v době návrhu ručně. Nám vyhovuje druhá možnost, proto ve vlastnostech bloku zvolíme *Set Initial Configuration*. Nyní se nám objeví jakýsi primitivní editor prvků, které může okno obsahovat. My si zatím přidáme pouze několik prvků typu *Label*, které si vhodně pojmenujeme. Nesmíme také zapomenout zaškrtnout políčko *Visible*, aby se okno zobrazilo již při startu aplikace.

  Nyní přivedeme notifikační výstup naší aktivity na vstup dialogu a nastavíme příslušné parametry. Budeme muset přepnout na ruční konfiguraci a vyplnit vždy název pole (v uvozovkách) a jeho hodnotu do pole *Text*.[75] Tím máme zajištěno, že souběžně s prováděnou kalibrací budeme informováni o jejím průběhu.

  Pokud teď aplikaci spustíme, objeví se okno, robot provede obě otočky, ale žádnou hodnotu neuvidíme. Zapomněli jsme totiž povolit kalibraci nastavením proměnné *Kalibrovat*. Zkusme pouze dočasně v inicializační akci *Start* nastavit hodnotu proměnné na *true* a spustit aplikaci znovu. Tentokrát už bychom mohli být spokojeni, data se aktualizují přesně podle plánu.

[75] Některé prvky naopak požadují hodnotu v parametru *Value*, podrobnosti najdeme v dokumentaci MRDS.

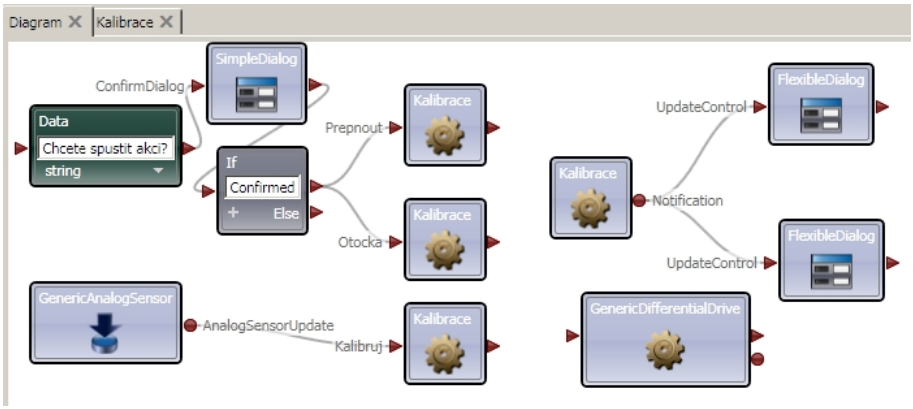


Obrázek 51: VPL_SledujCaruSKalibraci – upravená akce *Otocka* aktivity *Kalibrace*

– **Úprava 3 – povolení kalibrace**
Nyní tedy vraťme inicializaci do původního stavu a pokusme se řešit povolení kalibrace za běhu. Můžeme se o to pokusit tím, že tuto proměnnou nastavíme současně s příchodem požadavku na otočku a naopak ji vypneme současně s dokončením otočky. Přidejme tedy do akce *Otočka* příslušné bloky *Data* a *Variable* (takto upravenou akci vidíme na obr. *VPLSledujCaruSKalibraci-KalibraceOtocka2* a zkusme aplikaci spustit nyní. Vidíme, že robot se otáčí, ale žádné informace v okně nevidíme.Teprve v okamžiku, kdy robot zastaví, naskočí všechny hodnoty.

Tady jsme narazili na další omezení jazyka VPL. Protože všechny proměnné jsou součástí vnitřního stavu služby, jak jsme si už jednou vysvětlili (viz kapitola 5.2.6), platí i pro služby VPL stejná pravidla jako pro běžné DSS služby psané třeba v C#. Pokud se znovu podíváme na námi vytvářenou službu pro převod znaků v kapitole 4.8, zjistíme, že obslužné metody (handlery), které manipulují s proměnnými vnitřního stavu, jsou označeny jako *ServiceHandlerBehavior.Exclusive*. To jim zajišťuje exkluzivní, tedy výhradní přístup k proměnným, ostatní vlákna čekají.

Proto i ve VPL platí přísná omezení na použití proměnných. Vždy platí, že může běžet pouze jedna služba (její vlákno), která pracuje s proměnnými, ostatní čekají ve frontě. Naopak služby, které k proměnným nepřistupují, mohou běžet bez omezení. Podrobně je tato problematika popsána v sekci VPL Lab 5 – Exclusivity test dokumentace MRDS. Pro nás je důležité, že jak akce *Otocka*, tak *Kalibruj* s proměnnými pracují, proto nemohou běžet souběžně tak, jak bychom si přáli. Nejprve dokončí robot otočku a teprve poté se zpracují zprávy čekající zatím ve frontě na akci Kalibruj. Na výsledku se sice nic nemění, data byla nakonec zpracována, ale už třeba to, že jsme neviděli průběh tohoto



Obrázek 52: VPL_SledujCaruSKalibraci – diagram během ladění kalibrace

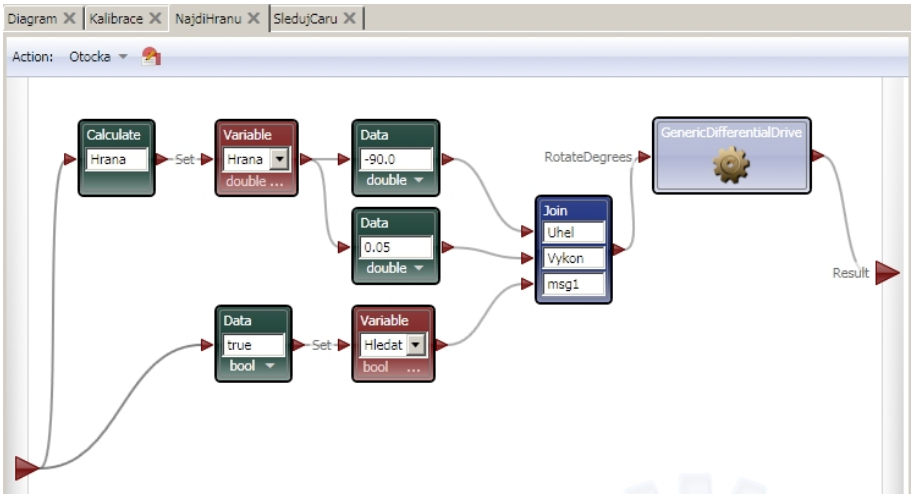
zpracování, může být u jiných aplikací na škodu. Je tedy třeba se takových postupů vyvarovat.

Zkusme tedy použít jiný přístup. Protože nemůžeme nastavit proměnnou přímo v akci *Otocka*, musíme to udělat někde jinde. Jediné místo, kde se o dokončení otočky dozvíme, je hlavní diagram, protože výstupní zpráva z portu aktivity dorazí právě po dokončení otočky. Teď pouze potřebujeme zvenku nastavit proměnnou *Kalibrovat*. Opět můžeme využít nové akce, tentokrát ji nazveme *Prepnout* a jediné, co bude dělat je to, že podle předaného parametru nastaví hodnotu proměnné. Takto jednoduchou akci si na obrázku ukazovat nemusíme, naopak se podívejme (obr. 52), jak v tuto chvíli vypadá hlavní diagram. Nyní už bude kalibrace fungovat správně.

- **Vyhledání hrany dráhy**
  Dalším postupným krokem je dosáhnout toho, aby se robot vrátil do místa, kde senzor hlásil přibližně střední hodnoty, tedy pravděpodobně na rozhraní dráhy a okolí. Proto musíme spustit další otočku opačným směrem, než byla ta poslední, a současně vyhodnocovat hodnoty senzoru.

  – **Sestavení diagramu**
    Také potřebujeme znát požadovanou střední hodnotu. Jediné místo, kde ji zatím můžeme získat, je aktivita *Kalibrace*. Vytvoříme tedy další akci, nazvanou *VratVysledek* a v ní tři parametry odpovídající třem vnitřním proměnným. Jediné, co tato akce bude obsahovat, budou bloky *Calculate* pro každou proměnnou, které přes *Join* přivedeme na výstup akce a správně spárujeme hodnoty parametrů.

    Nyní bychom mohli vytvořit v hlavním diagramu také vytvořit příslušné proměnné a zkalibrované hodnoty načíst, ale výhodnější bude je předat tam, kde najdou uplatnění, tedy přímo v aktivitě, která bude hranu vyhledávat.



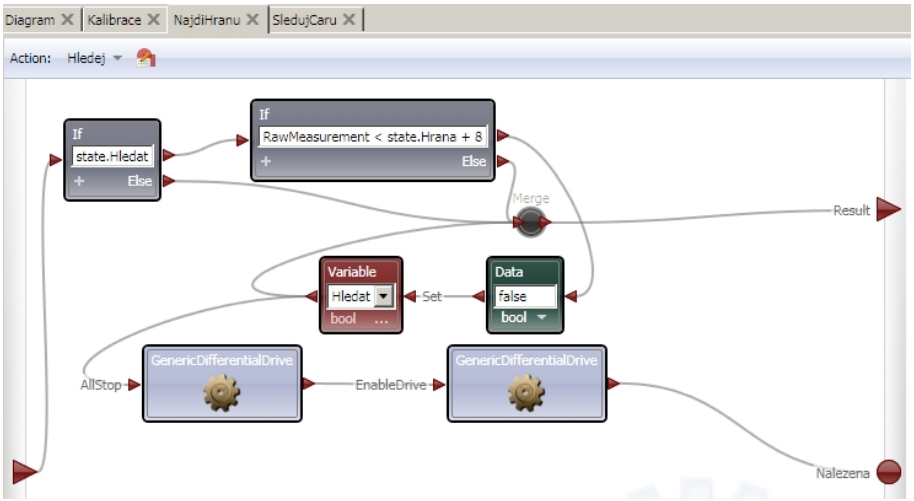
Obrázek 53: VPL_SledujCaruSKalibraci – akce *Otocka* aktivity *NajdiHranu*

Vytvoříme si tedy novou aktivitu nazvanou *NajdiHranu*. Ta bude mít dva základní úkoly, vykonat zpětnou otočku a zastavit při nalezení hrany. Budeme tedy potřebovat znát hraniční hodnotu osvětlení, minimum a maximum nás nezajímají. Proto si vytvoříme příslušnou proměnnou a rovnou také přepínač, podobně jako u kalibrace (také tady potřebujeme filtrovat údaje od senzoru v době, kdy nemá aktivita hranu hledat). Obě proměnné inicializujeme v akci *Start*.

Hlavní výkonnou akci si nazvěme třeba opět *Otočka*. Běžným způsobem v ní provedeme otočku o 90° zpět (abychom měli jistotu, že hraniční bod nalezneme). Teoreticky bychom měli i tady vytvořit další akci pro nastavení vstupních hodnot, ale je zde jeden podstatný rozdíl. Sice budeme skenovat terén během přesunu, ale už nebudeme v akci otočka čekat na její dokončení, protože konec operace už není závislý na dokončení otočky, ale na nalezení hrany. Proto můžeme nechat robota provést otočku a okamžitě akci ukončit. Nebudeme tedy potřebovat, aby tato akce běžela paralelně se sledováním senzoru, proto můžeme proměnné nastavit přímo v ní.

Protože nejrychlejší cestou, jak zjistit hodnotu hrany, bude nechat ji předat přímo z aktivity *Kalibrace*, vytvoříme v akci rovnou vstupní proměnnou, opět nejlépe nazvanou *Hrana*. Sestavení vlastního algoritmu otočky by už nemělo být složité, jeho aktuální stav vidíme na obrázku 53.

Dále potřebujeme provádět sledování údajů ze senzoru. Opět, stejně jako u kalibrace, budeme muset data přijímat zvenku, proto vytvoříme další akci (*Sleduj*) a v ní vstupní proměnnou *RawMeasurement*. Vlastní algoritmus musí nejprve vyhodnotit podmínku, zda smí vyhodnocovat a následně porovná naměřenou hodnotu s hodnotou uloženou ve své proměnné *Hodnota*. Pokud se vejde



Obrázek 54: VPL_SledujCaruSKalibraci – akce *Hledej* aktivity *NajdiHranu*

do přednastavené tolerance (tu stanovíme experimentálně), zastaví motor a vypne další hledání.

Protože nechceme nechat pro další aktivitu v aplikaci motor zablokovaný, raději do naší akce přidáme i opětovné povolení motoru nastavením *EnableDrive*.

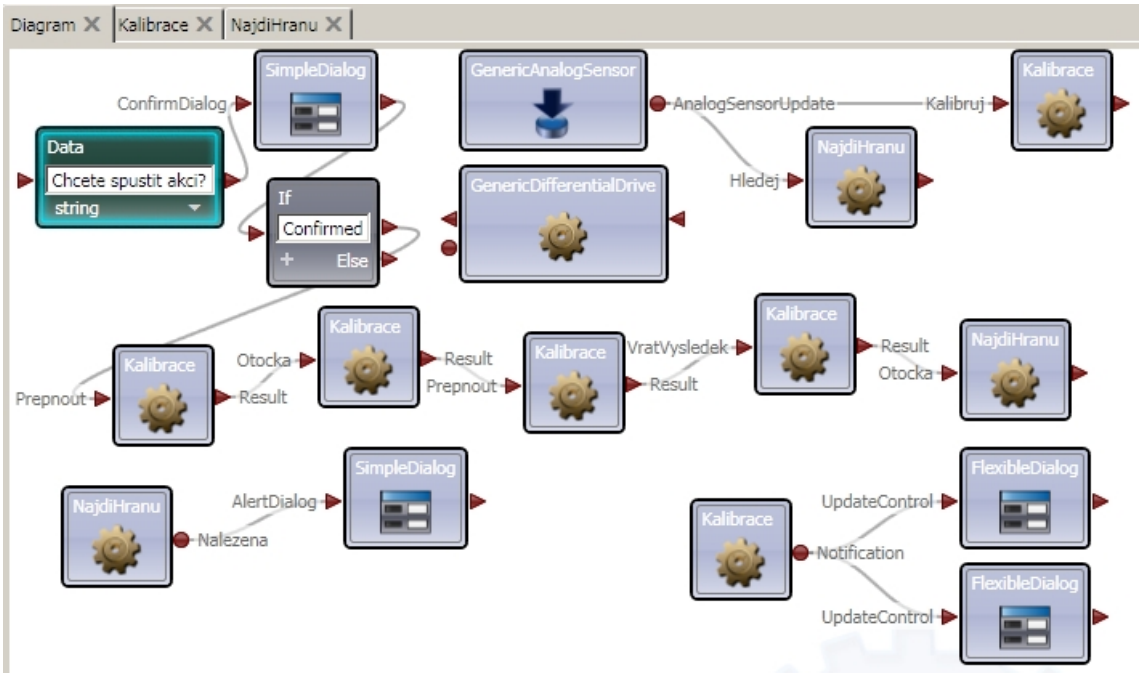
Teď si ovšem musíme rozmyslet, jak zařídit výstup z akce. Vzhledem k tomu, že je na vstupu navázána na notifikace senzoru, zřejmě by nemělo valný smysl přivést výstup na výstupní port, neměli bychom zprávu komu předat, protože mimo notifikací se na vstupní port nikdo jiný nepřipojí, aby očekával odpověď.

Lepším řešením tedy bude použít notifikace. To už jsme si vyzkoušeli u kalibrace pro výstup ladicích informací, proto postup pouze zopakujeme, tentokrát ani nebudeme potřebovat předávat žádné proměnné směrem ven. Notifikaci si nazvěme třeba *Nalezena*. Výsledný diagram je na obr. 54.

Poznámka 7.22 Obecně je dobrým zvykem použít notifikace všude tam, kde výstup přímo nesouvisí se vstupem, tzn., je lepší na výstupu pouze potvrdit přijetí nějakého požadavku, ale o dokončení operace informovat notifikací.

– **Spuštění**
Abychom mohli ověřit, že všechno funguje podle očekávání, vytvoříme si v hlavním diagramu jednoduchý dialog, který se zobrazí na základě notifikace od nové aktivity. Stav diagramu v této fázi vývoje je na obrázku 55.

- **Sledování dráhy**
Pro sledování dráhy použijeme mírně obměněný algoritmus z úlohy 7.15. Nebudeme korekce provádět vždy stejné, ale poměrně podle odchylky od hrany dráhy.



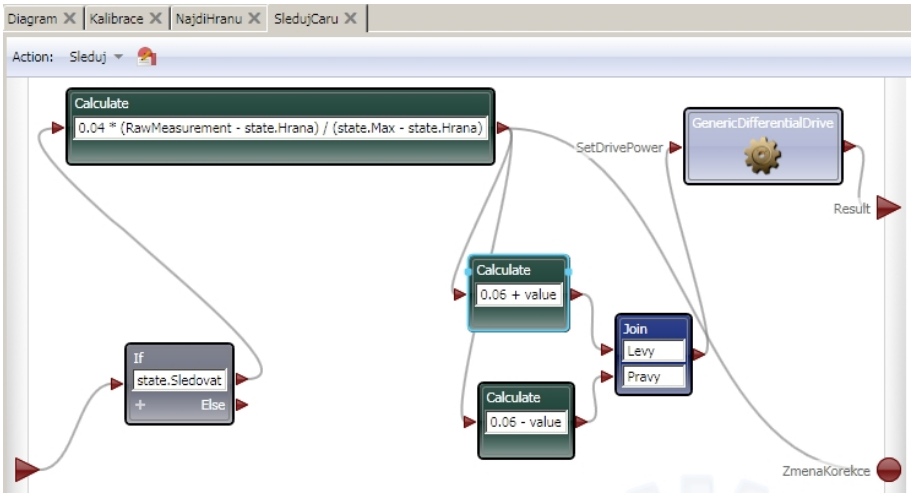
Obrázek 55: VPL_SledujCaruSKalibraci – hlavní diagram ve fázi hledání hrany

– **Sestavení diagramu**
  Abychom zachovali jednotnou formu aplikace, vytvoříme si i pro tuto operaci novou aktivitu nazvanou *SledujCaru* a v ní akci *Sleduj*. Ta by, jako už tradičně, měla mít vstupní proměnnou *RawMeasurement*.

  Když si projdeme jednotlivé fáze, které jsme zatím vytvořili, zjistíme, že už po dokončení kalibrace máme k dispozici hodnoty, které nová aktivita potřebuje, můžeme je tedy rovnou předat. Vytvoříme proto další akci, nazvanou třeba *Nastav* a v ní převezmeme předané parametry a nastavíme do příslušných proměnných (opět potom v akci *Start* provedeme jejich inicializaci).

  Jak tedy bude výpočet korekce probíhat ?[76] Zjistíme odchylku naměřené hodnoty od hodnoty hraniční (střední) a tu porovnáme s maximální možnou výchylkou (tedy rozdílem mezi maximální a hraniční). Budeme přitom uvažovat i znaménko, abychom získali i směr korekce. Z této úvahy dokonce zjistíme, že ani nepotřebujeme znát hodnotu minimální a nemusíme ji tedy vůbec přebírat. Pokud jsme to už udělali, můžeme ji nyní z proměnných aktivity odstranit. Takto získanou korekci bychom ještě měli násobit nějakou konstantou, která udá odpovídající změnu výkonu motoru. Hodnotu nejlépe zjistíme porovnáním s experimentálně zjištěnými hodnotami z první úlohy (viz 7.15), protože půjde o polovinu maximálního rozdílu mezi levým a pravým motorem. Výslednou korekci pak přičteme ke střednímu výkonu motoru pro levé kolo a

[76] Není to jediná možnost, můžeme použít i nelineární výpočet, kombinaci s vybočováním z dráhy apod.



Obrázek 56: VPL_SledujCaruSKalibraci – akce *Sleduj* aktivity *SledujCaru*

odečteme pro pravé kolo. Tím při maximální odchylce dosáhneme stejného manévru jako ve zmíněné první úloze.

Poznámka 7.23 Musíme dát pozor na správnou polaritu odchylky, jinak robot bude reagovat přesně opačně, což z jeho chování nemusí být úplně zřejmé a budeme obtížně hledat chybu.

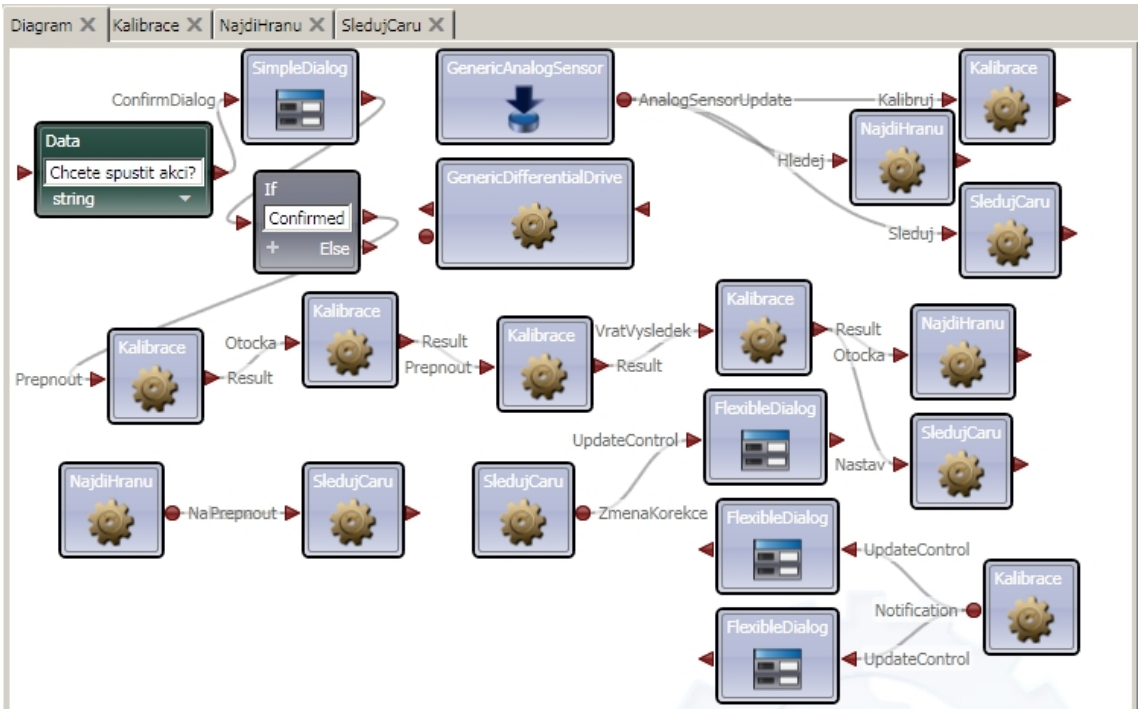
Takto připravená akce je na obrázku 56. Za pozornost stojí použití tří bloků *Calculate*, abychom se vyhnuli ukládání mezivýsledku do pomocné proměnné, a také využití notifikačního výstupu. Ten bude sloužit opět pro ladění a budeme jím hlásit aktuální hodnotu korekce. Tu pak zobrazíme také na *Flexible Dialogu* v nově vytvořeném prvku *Label*.

– **Úprava 1 – povolení notifikací**
Aktivita je tedy připravená, ale ještě musíme v pravý okamžik povolit její spuštění. To zajistíme právě notifikací od bloku *NajdiHranu*, proto si vytvoříme další akci, tentokrát nazvanou *Prepnout* a v té povolíme sledování.

- **Sestavení diagramu celé aplikace**
Hlavní diagram nám pomalu vyrůstal pod rukama již při začleňování jednotlivých dílčích aktivit, proto v této fázi můžeme pouze popropojovat příslušné datové toky mezi aktivitami, nastavit (zkontrolovat) správné sladění jednotlivých typů zpráv a vybrat odpovídající manifesty pro GDD a optický senzor.

- **Spuštění**
Výsledek by měl odpovídat našim očekáváním, byť bude ještě potřeba doladit některé hodnoty zadané napevno. To už ale opět ponechme experimentování.

  Když si prohlédneme výsledný diagram (obr. 57), bude vypadat daleko přehledněji, než diagramy minulých úloh. Pokud totiž dobře navrhneme vnitřní strukturu



Obrázek 57: VPL_SledujCaruSKalibraci – výsledný hlavní diagram

jednotlivých aktivit, stává se z vytvoření aplikace již jenom záležitost propojování příslušných portů ve správném pořadí, maximálně s nějakými podmínkami nebo větvením.

- **Námět na vylepšení**
  Aby byla aplikace opravdu reálně použitelná, třeba i po kompilaci aktivit na služby, bylo by vhodné vynést z ní prostřednictvím inicializačních akcí do hlavního diagramu všechny konstanty, tedy napevno zadané hodnoty. Jinak by se z nich stala tzv. „magická čísla“, což je sice oblíbená praxe amatérských programátorů, ale mezi odborníky si rozhodně uznání nezíská.

7.18 Úloha 14 – Ovládání třetího motoru – čelistí

7.18.1 Zadání

Podle pokynů uživatele otevírat a zavírat čelisti robota.

7.18.2 Postup řešení

- **Analýza**
  Model robota Tribot ve variantě, která je sestavena podle plánku dodaného výrobcem ke stavebnici, se shoduje s modelem, který máme k dispozici ve virtuálním prostředí SimplySim. Zásadní odlišností proti modelu ve výchozí simulaci MRDS je třetí motor, zde ve funkci řízení předních čelistí (klepet) robota. Přes jednoduchý převod jsou klepeta otevírána a zavírána podle směru otáčení motoru. Tuto úlohu tedy můžeme otestovat opět pouze v jednom virtuálním prostředí, případně na reálném robotovi.

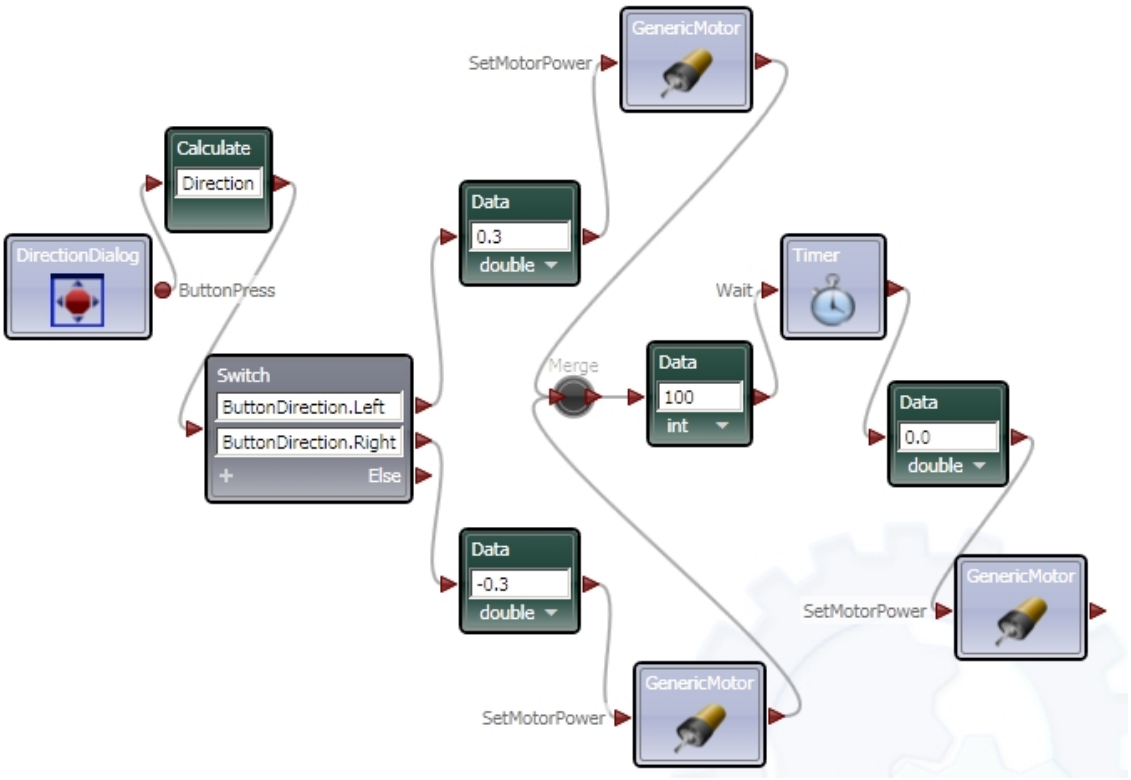
- **Použité služby**
  Tato úloha nepožaduje pohyb robota po podložce, proto tentokrát blok GDD nevyužijme. Stejně tak nepotřebujeme žádná čidla. Naopak poprvé využijeme generické služby *Generic Motor* pro ovládání třetího motoru robota.

  Dalším novým prvkem, který použijeme, bude jednoduchý ovládací panel se čtyřmi směrovými šipkami, který je součástí předdefinovaných služeb MRDS. Najdeme ho mezi ostatními službami pod názvem *Direction Dialog* a nevyžaduje žádnou úvodní konfiguraci.

- **Sestavení diagramu**
  V této úloze budeme ovládat statického robota, nemusíme tedy vkládat vstupní podmínku. Musíme si však vysvětlit, jak lze použít ovládací dialog. V dokumentaci MRDS je uveden příklad (VPL Tutorial 4), ve kterém je notifikace *ButtonPress* filtrována blokem *Switch* na základě textového řetězce. My použijeme podobnou metodu, ale místo textových řetězců zadáme konstantní hodnoty, které nám editor VPL nabídne. Nemusíme se tak bát případných překlepů a současně máme jistotu, že testujeme pouze ty hodnoty, které jsou opravdu k dispozici.

  Protože budeme čelisti pouze rozevírat a zavírat, stačí nám testovat pouze dvě tlačítka, zvolíme tedy *Left* a *Right*. Následně vložíme blok *Generic Motor*, kterému přiřadíme správný manifest a použijeme jeho akci *SetMotorPower*. Tento blok příliš funkcí nenabízí, proto musíme délku pohybu nastavit časovačem. Po uplynutí vhodné doby (závisí na rychlosti pohybu, kterou nastavíme) motor zastavíme. Tím zajistíme postupné rozevírání a zavírání čelistí podle počtu stisknutí příslušného tlačítka.

- **Spuštění**
  Hotový diagram vidíme na obrázku 58. Po spuštění se objeví dialogové okno se šipkami a klikáním postupně měníme stav čelistí robota.



Obrázek 58: VPL_PohybCelistiC – výsledný diagram

- **Námět na vylepšení**
  Pokud čelisti rozevřeme úplně, motor již běží „přes zuby“ převodu a zbytečně spotřebovává energii, podobně při čelistech úplně zavřených. Bohužel generický motor nemá možnost detekce otáček, ale zařazením počítadla můžeme alespoň částečně omezit zbytečnou aktivaci motoru v mezní poloze.

7.19 Úloha 15 – Zobrazení textu na Kostce NXT

V následujících úlohách si ukážeme využití služeb, které jsou specifické pouze pro robota ze stavebnice LEGO® MINDSTORMS® NXT a které nemáme zatím možnost vyzkoušet ve virtuálním prostředí.

7.19.1 Zadání

Zobrazit uživateli vstupní dialog s možností zadání textu. Robot poté dvakrát pípne a text zobrazí na displeji Kostky.

7.19.2 Postup řešení

- **Analýza**
  Pro tuto úlohu si vypůjčíme hotovou ukázkovou aplikaci, kterou najdeme v podadresáři *samples\VplExamples\LEGO\MsgReadWrite* instalace MRDS.[77] Ta demonstruje spolupráci aplikací psaných ve VPL s nativními aplikacemi psanými v NXT-G a běžícími přímo v robotu. V tomto případě prostřednictvím Bluetooth pošleme do Kostky zprávu, kterou nativní program běžící v Kostce přijme a zobrazí na displeji.

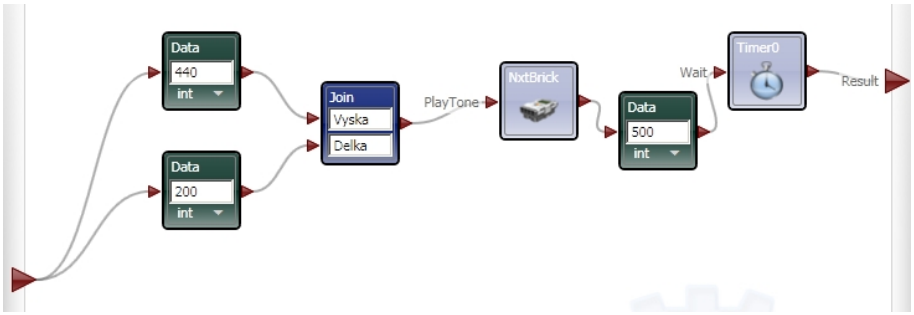
- **Použité služby**
  Tentokrát poprvé budeme používat negenerické služby určené přímo pro robota LEGO® MINDSTORMS® NXT. Zde to bude blok *Lego NXT Brick (v2)* pro konfiguraci Kostky a *Lego NXT Block I/O (v2)* pro obsluhu V/V operací v Kostce. Dále se využívají služby *Text functions* pro operace s textovými řetězci a *Sound Player* pro potvrzující pípnutí v PC. Pro implementaci pípání na straně robota pak použijeme i blok *Timer*.

- **Sestavení diagramu**
  Úloha využívá diagram ze stávající ukázkové úlohy, který si můžeme přímo otevřít. Z důvodu pozdější editace ale bude lepší vytvořit jeho kopii a pracovat s ní.

- **Spuštění**
  V adresáři ukázkové aplikace najdeme také soubor *ProcessMessage.rbt*. Přípona *.rbt* je typická pro nativní aplikace psané v NXT-G. Proto musíme použít originální programovací prostředí (dodané se stavebnicí) a tento soubor v editoru otevřít, zkompilovat a odeslat do Kostky. Postup této činnosti je intuitivní, přesto lze v případě pochyb nahlédnout i do přibalené příručky.

  Následně spustíme naši aplikaci ve VPL. Pro přiblížení činnosti algoritmu odkážeme na dokumentaci k MRDS, kde je celá ukázka popsána, zde nám postačí vědět, že nejprve zkontroluje přítomnost požadovaného programu, ten následně spustí (pokud ještě spuštěn není) a předá mu uživatelem zadaný text. Program v Kostce text zobrazí na displeji a vrátí zpět naší aplikaci.

[77] Kopie ukázkové aplikace s doplněními dle tohoto zadání je uložena i na CD, jež je součástí této práce, pod názvem *VPL_ZobrazZpravu* a vztahují se na ni stejná autorská práva jako na její originál.

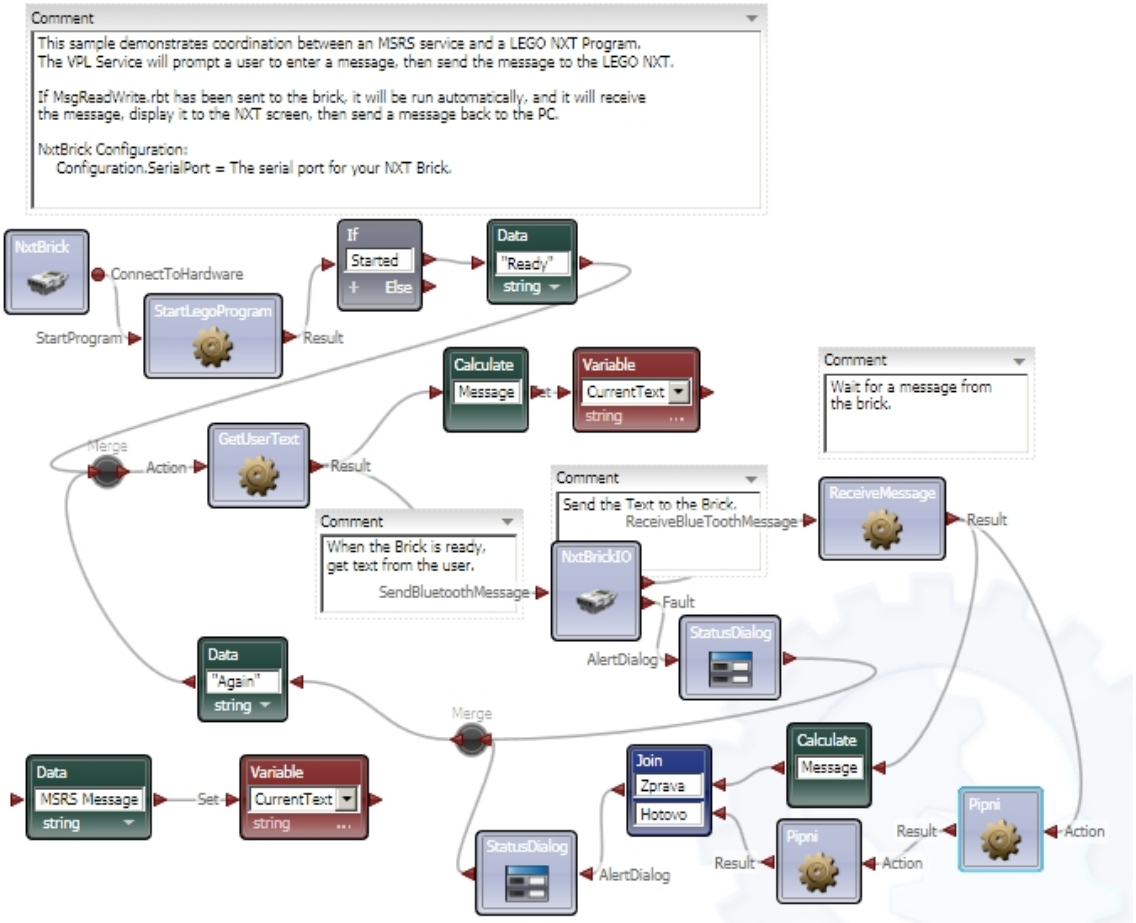


Obrázek 59: VPL_ZobrazZpravu aktivita *Pipni*

- **Úprava**
  Proti původní úloze přidáme ještě požadované zvukové efekty na straně Kostky. Za tímto účelem vložíme *Timer* a kopii bloku *NxtBrick*. Využijeme přitom jeho akci *PlayTone*, která jako parametr požaduje frekvenci a délku tónu. Pošleme tedy nejprve jeden tón a po uplynutí krátkého času (vypršení *Timeru*) pošleme tón znovu. Protože hlavní diagram je sestaven z několika uživatelsky definovaných aktivit, dodržíme stejnou formu a pro pípnutí si vytvoříme vlastní (viz obr.59.

  Následně zařadíme blok dvou těchto aktivit do původního diagramu. Abychom zachovali zobrazení vráceného textu v dialogu, vedeme zprávu potvrzující provedení úkolu další větví a po pípnutích přivedeme na vstup dialogu. Takto sestavený diagram vidíme na obr. 60.[78]

[78] Ostatní aktivity zůstávají beze změny, proto je zde neukazujeme.



Obrázek 60: VPL_ZobrazZpravu – výsledný diagram

7.20 Úloha 16 – Zobrazení textu se zařazením vlastní služby

7.20.1 Zadání

Obdoba předchozí úlohy s tím, že využijeme službu napsanou přímo v C#.

7.20.2 Postup řešení

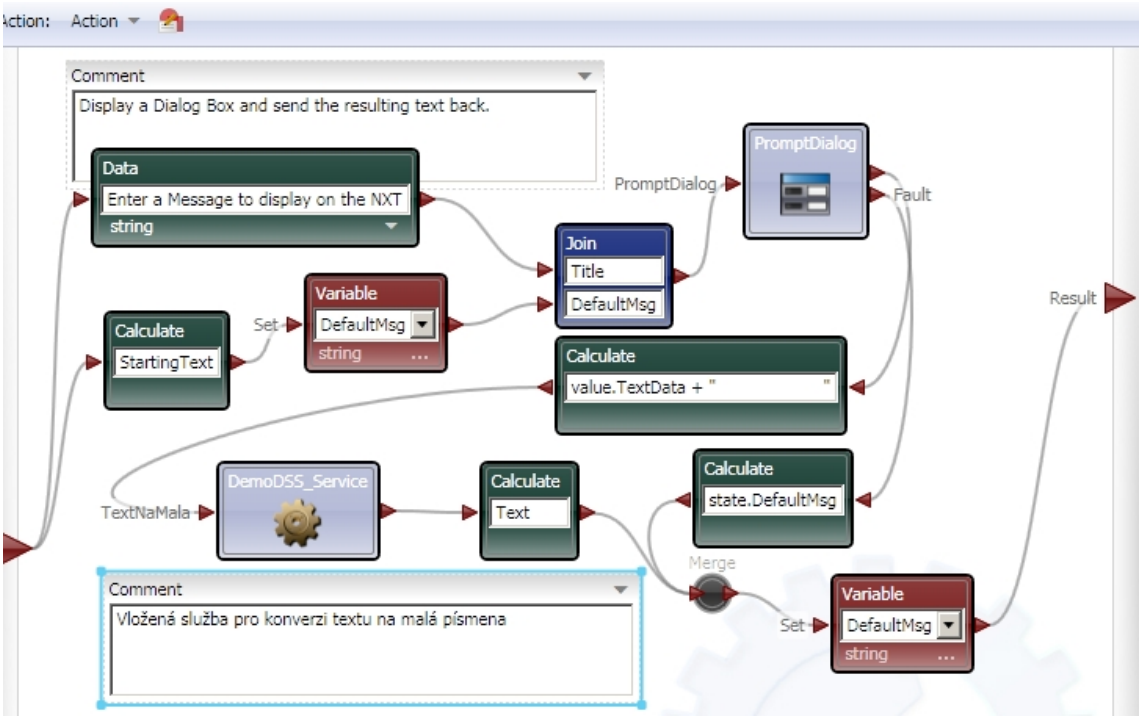
- **Analýza**
  Předchozí úlohu jsme zde uváděli zejména proto, abychom mohli nyní přistoupit k zařazení naší vlastní služby napsané v jazyku C#. Tu jsme si podrobně představili v kapitole 4.8 a měla velmi jednoduchý úkol, převod všech znaků v zaslaném řetězci na malá resp. velká písmena. Nyní tuto službu, kterou bychom nyní měli vidět mezi ostatními službami v levém spodním okně editoru VPL, můžeme do naší aplikace zařadit.

- **Použité služby**
  Použijeme všechny služby z předchozí úlohy a navíc samozřejmě námi vytvořenou službu *DemoDSS_Service*.

- **Sestavení diagramu**
  Naši službu můžeme umístit do aktivity *GetUserText* a zařadit ji mezi vstupní pole s uživatelem zadaným textem a blok *Merge*. Současně vybereme typ požadované operace, zda půjde o převod na malá, nebo velká písmena. Tuto aktivitu vidíme na obr. 61.

- **Spuštění**
  Základní diagram může zůstat stejný jako u předchozí úlohy, proto takto upravenou aplikaci můžeme spustit a okamžitě bychom měli vidět požadovaný výsledek. Službu není třeba nijak konfigurovat a o její činnosti se můžeme přesvědčit i zobrazením jejího stavu v internetovém prohlížeči.

  Poznámka 7.24 Takto jednoduše můžeme do aplikace integrovat libovolnou zkompilovanou službu DSS. Může se jednat o službu speciálně vytvořenou v C# (či jiném jazyce pro .NET), ale i některou aktivitu původně psanou ve VPL. Tu pak můžeme zkompilovat a tímto způsobem použít, případně provést její export do C#, kód dále upravit a po kompilaci opět použít jako samostatný blok ve VPL.



Obrázek 61: VPL_FiltrujAZobrazZpravu – aktivita *GetUserText*

7.21 Úloha 17 – Využití detektoru zvuku

7.21.1 Zadání

Robot se vydá vpřed a při detekci hlasitého zvuku začne couvat.

7.21.2 Postup řešení

- **Analýza**
  Tato úloha je evidentně opět variací na jednu z prvních úloh, která požadovala couvání robota při kontaktu s překážkou (kapitola 7.11). Místo tlaku (bumper) tu ale reagujeme na zvuk. Můžeme tedy s výhodou využít kostru původní úlohy a změnit pouze nezbytné části.

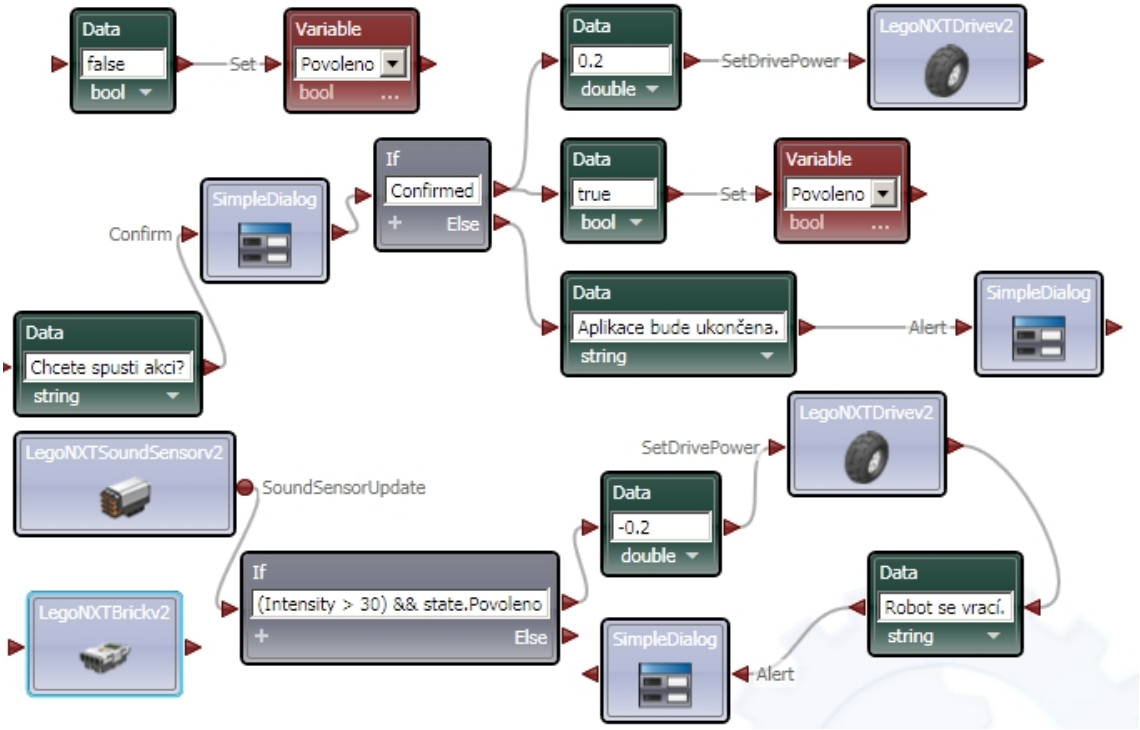
- **Použité služby**
  Pro tuto úlohu využijeme další senzor, který není součástí virtuálního prostředí a pro odzkoušení aplikace tak musíme použít reálného robota. Na plochu diagramu tedy vložíme blok *Lego NXT Sound Sensor (v2)* a vyměníme generické služby (GDD) za jejich ekvivalent z nabídky LEGO NXT, tedy *Lego NXT Brick (v2)* a *Lego NXT Drive (v2)*. Všechny služby musíme nastavit volbou *Set initial configuration* a vyplněním potřebných hodnot. Pro nový senzor se jedná pouze o uvedení služby Kostky a čísla portu, do kterého je v Kostce připojen.

- **Sestavení diagramu**
  Pokud použijeme jako výchozí bod diagram z kapitoly 7.11 (Při nárazu couvej), bude stačit pouze výměna generické služby GDD za konkrétní. Typy a parametry zpráv můžeme ponechat beze změny, protože i tato konkrétní implementace služby odpovídá předpisu GDD. Místo výstupu bumperu pak zapojíme výstup zvukového senzoru, jehož úroveň vyhodnotíme v podmínce pro couvání robota.

- **Spuštění**
  Takto upravený diagram můžeme spustit a po úspěšném navázání spojení s Kostkou i ověřit jeho funkci. Zřejmě opět zjistíme, že i v tomto případě budeme muset rozhodovací úroveň senzoru upravit podle skutečně naměřených hodnot. Můžeme si opět pomoci dashboardem a případně i blokem *Flexible Dialog*, na kterém si můžeme nechat zobrazovat aktuálně přijaté hodnoty senzoru.

  Při pozornějším pohledu na diagram ale zjistíme, že data od detektoru přicházejí neustále a do motoru jsou tedy zasílány pokyny pro nastavení rychlosti při jakémkoli překročení nastavené úrovně akustického tlaku, ať už byla rychlost jednou nastavena, nebo ne. Rovněž nemusíme vůbec odsouhlasit vstupní dialog a robot se při hlasitějším zvuku sám rozjede vzad. V případě bumperu nám to nevadilo, náraz do překážky se předpokládá až po uvedení robota do pohybu, ale zde už je situace jiná.

  Podobně jsme řešili zamezení automatického rozjetí robota i při první úloze se sonarem (viz kapitola 7.12). Proto také zde je nutné vložit do testu další podmínku a



Obrázek 62: VPL_PriZvukuCouvni – výsledný diagram

vytvořit novou proměnnou, kterou povolíme příjem zpráv teprve po odsouhlasení uživatelem. Současně tuto proměnnou opět vypneme po první změně rychlosti robota.

Výsledný diagram vidíme na obr. 62.

- **Námět na vylepšení**
  Takto sestavený algoritmu reaguje pouze na první povel. Zajímavá by mohla být třeba taková modifikace, která by robota při každém hlasitém povelu otočila o 180°, zvýšila nebo snížila rychlost apod. V praxi by to sice nebylo příliš použitelné, ale lze si představit i variantu, kdy by se na základě časové podmínky vyhodnocoval nějaký konkrétní povel jako posloupnost zvuků.

7.22 Úloha 18 – Monitorování stavu baterie

7.22.1 Zadání

Robot podává informace o stavu své baterie a při poklesu napětí pod určitou úroveň zastaví.

7.22.2 Postup řešení

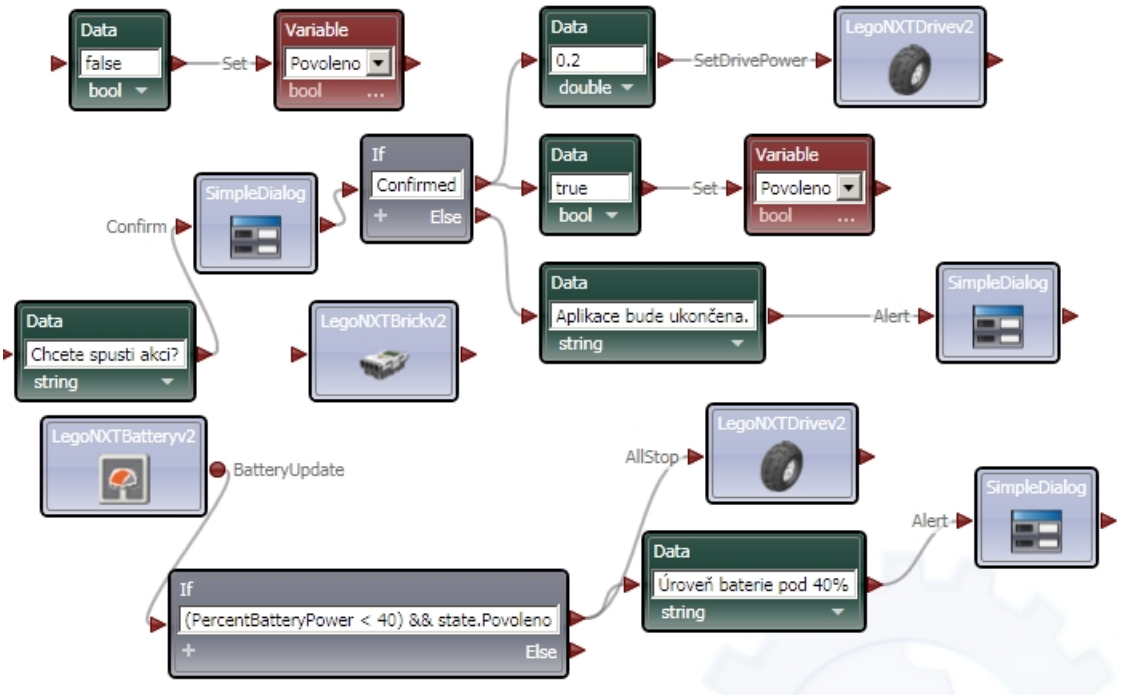
- **Analýza**
  Při experimentech s reálným robotem si velmi rychle všimneme, že jeho slabinou jsou právě baterie. Zejména u mobilních aplikací, kdy jsou v provozu motory robota, je jejich výdrž opravdu hodně malá a pohybuje se v řádu minut. Jednou z funkcí Kostky je i monitorování stavu baterie. Pro její demonstraci můžeme využít v podstatě libovolnou z předchozích úloh, ovšem vzhledem k tomu, že informace o baterii nejsou dostupné v simulacích, musíme tentokrát použít reálného robota.

- **Použité služby**
  Blokem, který použijeme poprvé, je *Lego NXT Battery (v2)*. Samozřejmě opět musíme správně nastavit spojení se službou *Lego NXT Brick (v2)*.

- **Sestavení diagramu**
  Pokud využijeme některý z diagramů z předchozích úloh (např. zastavení při nárazu do překážky), bude stačit přidat pouze jednu nezávislou část. Ta bude přijímat notifikace bloku baterie (*BatteryUpdate*) a na základě hodnoty *PercentBatteryPower* případně zastaví motory a zobrazí upozornění pro uživatele. Tento diagram je na obrázku 63.

- **Spuštění**
  Na této aplikaci není příliš co ladit, musíme ovšem správně nastavit hraniční hodnotu. Příliš malou nám asi Kostka vůbec neoznámí, protože už nebude mít dost energie ani na napájení Bluetooth adaptéru, příliš velkou zase možná nebudeme mít k dispozici už při spuštění aplikace. Můžeme ovšem ve fázi ladění nechat zobrazovat přijaté hodnoty a hranici stanovit podle nich.

- **Námět na vylepšení**
  Blok baterie nabízí ještě další typ notifikace (*CriticalLevelUpdate*). Ta je zaslána teprve v případě, že stav baterie klesne pod předem nastavenou kritickou úroveň. Tu nastavíme už při návrhu v konfiguraci služby v pravém dolním rohu editoru nebo za běhu programu zasláním příslušné zprávy. Pokud nás tedy nezajímají průběžné hodnoty stavu baterie, ušetříme cenný strojový čas a komunikační pásmo, když odchytíme pouze tuto notifikaci.



Obrázek 63: VPL_PriPoklesuBaterieZastav – výsledný diagram

7.23 Rozdíly proti NXT-G

Přestože nativní jazyk NXT-G není předmětem této práce a zájemce se s ním musí seznámit sám (lze doporučit třeba [27]), uveďme alespoň některé zásadní rozdíly ve srovnání s jazykem VPL.

7.23.1 Aplikace běží mimo Kostku

V úloze, která se zabývala zobrazením textu na displeji Kostky (7.19), jsme první rozdíl mohli zaznamenat. Nativní aplikaci musíme nejprve zkompilovat, nahrát do Kostky a teprve v ní spustit. Pro vlastní běh aplikace (provádění úlohy robotem) už není vývojové prostředí potřeba, programový kód běží přímo v procesoru Kostky. Robot je tak zcela autonomní.

Výhodou tohoto přístupu je právě naprostá nezávislost na jiném HW a přítomnosti či kvalitě spojení, nevýhodou pak nároky na výkon procesoru robota, který musí veškeré úkony provádět sám a v reálném čase.[79] Z hlediska programátora pak určitě největší nevýhodu představuje nutnost ukládání nového kódu do Kostky po každé změně ve vývojovém prostředí a nutnost „ostrých" testů přímo robotem. To s sebou nese v případě Lego® NXT dále nutnost častější výměny baterií, zvýšené namáhání dílů, v raném stádiu vývoje třeba i jejich zničení (pád ze schodů), apod.

Naproti tomu MRDS volí přístup zcela odlišný. Vlastní aplikace běží v reálném čase na PC, s robotem udržuje trvalé spojení, zjišťuje stavy jeho senzorů a řídí jeho pohyb předáváním příkazů jeho motorům. Výhody a nevýhody proti předchozímu modelu můžeme vlastně zrcadlově obrátit. V některých případech ovšem může být množství předávaných dat takové, že se kritickým místem může stát i šířka přenosového pásma mezi robotem a PC. Na misku vah ve prospěch MRDS pak můžeme přidat také možnost vizualizace (off-device testování) a vytváření aplikací pro více robotů.

Nemůžeme rozhodnout, který z výše uvedených přístupů je výhodnější, aniž bychom současně uvažovali účel vyvíjené aplikace a prostředí, ve kterém má být robot nasazen. Pokud vyžadujeme, aby byl robot opravdu naprosto nezávislý a mohl operovat i v prostředí bez spojení s řídícím centrem, je vhodným kandidátem na vývoj NXT-G. Ve většině ostatních případů ale budeme zřejmě volit MRDS.[80]

7.23.2 Aplikace řízené událostmi

V nativním vývojového prostředí NXT-G je na programátorovi, aby sám četl stavy jednotlivých senzorů a podle toho řídil běh aplikace. Nejčastěji se tedy tyto často se opakující procedury umisťují do cyklů, kdy jako podmínka slouží aktuální hodnota příslušného senzoru. Zjevnou nevýhodou tohoto postupu je, že programátor musí stále myslet na to,

[79] Existují speciální případy, kdy může robot spolupracovat přes Bluetooth s jinými roboty a teoreticky tak může jeden převzít úlohu „šéfa".

[80] Uvedená prostředí nepředstavují jediné možnosti. V kapitole 6.4 jsme zmínili alternativní FW a jazyky, z nichž každý má své výhody a uplatnění, ve kterých se nejvíce projeví. Samotné MRDS umožňuje spouštět aplikace v zařízeních, která podporují .NET, a to i ve verzi Compact Framework (CF), tedy např. Windows CE [21].

které senzory mohou v daném okamžiku změnit stav a které tedy musí v daném bloku kódu testovat. Pokud některé testovat zapomene, může to mít fatální následky pro běh aplikace (srážka, nezaznamenání průniku narušitele apod.). Pokud jich testuje naopak příliš mnoho a příliš často, neblaze se to projeví na efektivitě kódu [81] a zejména rychlosti jeho provádění. Některé rychlé děje tak aplikace nemusí vůbec zaznamenat, případně pouze s velkým zpožděním. Následky pak mohou být podobné jako v předchozím bodě.

Firma Microsoft proto ve svém vývojovém prostředí MRDS zvolila přístup, jež uvedené problémy řeší paralelním zpracováním operací, o což se stará běhové prostředí CCR, které jsme si dostatečně představili v kapitole 3. Částečně se tento model podobá řízení aplikací na základě tzv. událostí[82], ale jde mnohem dál v efektivitě a způsobu sestavení kódu. Připočteme-li ještě relativně jednoduchý jazyk VPL, který tvorbu kódu dále zjednodušuje až téměř na úroveň NXT-G, můžeme v MRDS vytvořit se stejnou námahou podstatně efektivnější kód než v nativním prostředí.

Skutečně paralelní zpracování více programů současně není na jednoprocesorovém sytému možné a v praxi se realizuje přidělováním procesorového času střídavě několika programům (procesům). Tomuto systému se říká multitasking [35] a jeho konkrétní implementace a řízení je věcí operačního systému, v našem případě MS Windows. Protože z předchozí části 7.23.1 víme, že v případě MRDS aplikace řídící NXT běží na PC, je vlastně pouze jedním z procesů, které na něm běží v témže okamžiku, a OS jí přiděluje procesorový čas jako celku.

O dělení procesorového času mezi části aplikace (zde procedury reagující na stav senzoru) se tedy musí postarat běhové prostředí, nad kterým aplikace běží. To umožňuje tzv. multithreading, současný běh více vláken. Detailní vysvětlení opět ponechme učebnicím příslušných programovacích jazyků [15], případně zájemce můžeme odkázat na [10], kde nalezne velmi názorný výklad paralelismu obecně. MRDS jde oproti samotnému prostředí .NET ještě dál a prostřednictvím svého nového běhového prostředí CCR Runtime[83] nabízí možnost snadného vytváření nejen paralelních, ale i asynchronních a distribuovaných aplikací. Blíže jsme tento princip vysvětlili v kapitolách 3 a 4, zde pouze poznamenejme, že jazyk VPL je na využití vlastností CCR a DSS založen a programátor tak nemusí věnovat pozornost např. synchronizaci vláken, dostupnosti dat, detailnímu ošetřování chyb apod.

7.23.3 Podpora více robotů současně

Zejména z části 7.23.1 plyne, že prostředí MRDS je velmi vhodné pro vývoj aplikací obsluhujících více než jednoho robota. Zatímco v NXT-G by taková aplikace sice byla možná, ale vlastní interakce mezi roboty by musela být implementována v každém z nich, zde tato podmínka odpadá.

[81] Toto je problém zejména vyšších programovacích jazyků (jako právě NXT-G), které před programátorem skryjí spoustu automaticky generovaného kódu.

[82] Podrobné vysvětlení tohoto pojmu najde zájemce v každé učebnici novějšího programovacího jazyka (např. [15, 34])

[83] Nejde o samostatné běhové prostředí (framework), ale o nadstavbu nad .NET.

Robot nemusí mít o existenci svých „kolegů“ ani tušení, pouze předává získaná data (veličiny snímané senzory) centrálnímu pracovišti (aplikace běžící na PC) a vykonává jeho pokyny (pohyb v terénu, úchop předmětů apod.). Pokud budeme uvažovat např. modelovou situaci družstva robotických fotbalistů, budeme potřebovat pouze jedenáct identických robotů[84] a jeden centrální počítač, který sbírá informace ze senzorů jednotlivých „fotbalistů“, vytváří si jejich kombinací virtuální mapu hřiště s pozicemi míče, vlastních i soupeřových hráčů, určuje podle aktuálního vývoje strategii hry a uděluje hráčům pokyny. Úspěšnost takového týmu bude patrně daleko vyšší ve srovnání s týmem naprosto autonomních robotů, z nichž každý si vytváří strategii vlastní.[85]

7.23.4 Distribuované aplikace

Z kapitoly 4 jsme se dozvěděli, že MRDS umožňuje rozdělit vykonávání kódu aplikace mezi více procesů na různých strojích. To dále zvyšuje výkon takovýchto aplikací a jejich robustnost. Při návrhu aplikace pro velmi sofistikovaného robota (nebo skupiny robotů) s množstvím senzorů a aktuátorů tak nejsme omezeni výpočetní silou jediného stroje, ale můžeme zátěž rozložit, a to i dynamicky podle aktuálních nároků.

7.23.5 Platformová nezávislost

Zatímco Lego® NXT-G je vývojové prostředí LabVIEW™ upravené speciálně pro stavebnici LEGO® MINDSTORMS® NXT a aplikace pro jiné roboty v něm vyvíjet nemůžeme, MRDS tuto možnost nabízí, dokonce je i hlavním důvodem jeho vzniku [1], a NXT zde tvoří pouze jednu z cílových platforem, kterou ovšem Microsoft zařadil již od první verze mezi ty, na kterých v dokumentaci a ukázkových aplikacích demonstruje funkce MRDS. Lze předpokládat, že tak učinil zejména z didaktických důvodů.

7.23.6 Programovací jazyk

Na rozdíl od NXT-G, což je grafický jazyk založený na modifikaci příkazů prostředí LabVIEW™, MRDS umožňuje vývoj aplikací do jisté míry nezávisle na použitém jazyku.

Preferovanou variantou je použití vizuálního jazyka VPL, ovšem některé části kódu (v extrémním případě i kód celý) lze psát v C#. Prostřednictvím Visual Studia lze pak volat příslušné procedury i z jiných jazyků.

Samotný jazyk VPL není omezený množinou již existujících příkazů, ale lze jej jednoduše doplňovat přidáváním nových služeb (services) nebo úpravami těch stávajících.[86]

[84] V praxi by se nejspíš minimálně brankář od ostatních mírně lišil

[85] Nebereme zde v potaz další podmínky, určující úspěch takového týmu, jako je např. dostatečný výkon PC, nezbytnost trvalého kvalitního spojení mezi PC a všemi hráči a nakonec také pravidla takového zápasu, která mohou určovat řadu omezení.

[86] Pro úplnost je třeba dodat, že toto lze i u NXT-G prostřednictvím originálního prostředí LabVIEW™.

8 Závěr

Rozsah této práce neumožňuje pokrýt danou problematiku v plné šíři, přesto autor věří, že poskytl dostatečně ucelený přehled a možná i vyvolal v některém ze čtenářů zájem o další studium tohoto rychle se rozvíjejícího oboru. Pokud se tak stalo, může mu jako další krok doporučit vynikající knihu [1], z níž sám čerpal první informace o tématu.

Zde uvedené úlohy demonstrovaly některé z typických postupů při vývoji aplikací ve VPL a zcela záměrně se autor snažil upozornit na co nejvíce problémů, se kterými se může tvůrce VPL aplikace setkat. Pro skutečné zvládnutí jazyka je však nezbytná vlastní praktická zkušenost včetně vlastních omylů a slepých cest, protože ty nejlépe donutí člověka o problému přemýšlet, hledat řešení, případně se věc podívat ze zcela nového úhlu. Také z tohoto důvodu byla v závěru několika úloh navržena další vylepšení jako inspirace pro vlastní pokusy.

Možným námětem na další práce na podobné téma je třeba již podrobnější zpracování vývoje aplikací (nejen robotických) v CCR z hlediska efektivity paralelního běhu aplikace, dále je možné rozpracovat omezení plynoucí z nasazení na platformách podporujících pouze .NET CF, vytvoření dalších virtuálních prostředí pro simulaci i jiných modelů robota ze stavebnice LEGO® MINDSTORMS® NXT, případně se zaměřit na tvorbu simulací v MRDS obecně.[87]

Jiří Melcr

[87] V této oblasti se před nedávnem objevil velmi nadějný pokrok v podobě skriptovacího jazyka SPL, viz [8].

9 Literatura

[1] JOHNS, Kyle, TAYLOR, Trevor, *Professional Microsoft® Robotics Developer Studio*, Indianapolis (Indiana): Wiley Publishing, Inc., 2008. ISBN 978-0-470-14107-6.

[2] TMEJ, Bohuslav Bc., *Ovládání laboratorního modelu robota Mindstorms NXT (machine robot) pomocí PC*, Zlín, 2008, Diplomová práce na Univerzitě Tomáše Bati ve Zlíně.

[3] DZIEKANIK, Martin, *Simulátor florbalu robotů vytvořený v Robotic studio*, Ostrava, 2008, Diplomová práce na Vyskoké škole báňské - Technické univerzitě Ostava na katedře informatiky, vedoucí diplomové práce Jan Martinovič.

[4] HORČIČÁK, Tomáš, *Fotbal robotů a Microsoft Robotic Studio*, Ostrava, 2009, Diplomová práce na Vyskoké škole báňské - Technické univerzitě Ostava na katedře informatiky, vedoucí diplomové práce Jan Martinovič.

[5] PASTRŇÁK, Jan, *Microsoft Robotics Studio - programování robotů*, Ostrava, 2008, Diplomová práce na Vyskoké škole báňské - Technické univerzitě Ostava na katedře informatiky, vedoucí diplomové práce Eliška Ochodková.

[6] *MSDN – Server firmy Microsoft pro vývojáře (sekce MRDS)* [online], URL:<http://msdn.microsoft.com/en-us/robotics/default.aspx>, [cit. 2010-05-03].

[7] *Microsoft Robotics Developer Center – fórum VPL* [online], URL:<http://social.msdn.microsoft.com/Forums/en-US/roboticsvisual-programminglanguage/threads>, [cit. 2010-01-03].

[8] *HelloApps – stránky o SPL (Simulation Programming Language)* [online], URL:<http://www.helloapps.com>, [cit. 2010-04-23].

[9] *Aplikace SimplySim ke stažení* [online], URL:<http://www.simplysim.net/DL/NXT-MSRDS-R2.exe>, [cit. 2010-01-03].

[10] HANÁK, Ján, *Základy paralelného programovania v jazyku C# 3.0.*, 1. vyd., Brno: Artax a.s., 2009, ISBN 978-80-87017-03-6.

[11] Channel 9, *Videopřednáška o CCR* [online], URL:<http://channel9.msdn.com/shows/Going+Deep/Concurrency-and-Coordination-Runtime>, [cit. 2009-12-18].

[12] *Microsoft CCR and DSS Toolkit 2008 R2* [online], URL:<http://www.microsoft.com/ccrdss>, [cit. 2010-05-03].

[13] *Vyjádření tvůrce MRDS k budoucnosti CCR* [online], URL:<http://social.msdn.microsoft.com/Forums/en-US/roboticsccr/thread/-de3f596a-e6fa-4431-b6ae-57afe1d051b1>, [cit. 2010-05-01].

[14] *CCR – Online User Guide* [online],
URL:<http://msdn.microsoft.com/en-us/library/bb905470(v=MSDN.10).aspx>,
[cit. 2010-05-03].

[15] BĚHÁLEK, Marek, *Podklady pro výuku jazyka C#* [online],
URL:<http://www.cs.vsb.cz/behalek/vyuka/pcsharp/text/index.html>,
[cit. 2010-05-03].

[16] *DSS – Online USer Guide* [online],
URL:<http://msdn.microsoft.com/en-us/library/bb905471(v=MSDN.10).aspx>,
[cit. 2010-05-03].

[17] *Arvindra Sehmi's Half-Baked Ideas and Digressions* [online],
URL:<http://blogs.msdn.com/asehmi/archive/2008/06/12/ccr-dss-use-cases-in-the-enterprise.aspx>, [cit. 2010-04-07].

[18] *DSSP – popis protokolu* [online],
URL:<http://purl.org/msrs/dssp.pdf>, [cit. 2010-05-03].

[19] *Microsoft Robotics Developer Center – fórum DSS služeb* [online],
URL:<http://social.msdn.microsoft.com/Forums/en-US/roboticsdss/threads>,
[cit. 2010-04-07].

[20] *Vyjádření tvůrce MRDS k notifikacím ve vlastních aktivitách* [online],
URL:<http://social.msdn.microsoft.com/Forums/en-US/roboticsvisualprogramminglanguage/thread/214c7122-a669-47da-a284-af7802199a25>,
[cit. 2010-05-03].

[21] MORGAN, Sara, *Programming Microsoft Robotics Studio*, Redmond (Washington): Microsoft Press, 2008, ISBN 0-7356-2432-1.

[22] KNUDSEN, Jonathan, *Lego MindStorms: Lego and MIT*,
URL:<http://www.oreillynet.com/pub/a/network/2000/01/31/mindstorms/index1b.html>, [cit. 2009-11-15].

[23] ASTOLFO, Dave, FERRARI, Mario, FERRARI, Giulio, *Building Robots with LEGO Mindstorms NXT*, Burlington (Massachusetts): Syngress Publishing, Inc., 2007, ISBN-13 9781597491525.

[24] GASPERI, Michael, HURBAIN, Philippe, HURBAIN, Isabelle, *Extreme NXT: Extending the LEGO MINDSTORMS NXT to the Next Level*, Berkeley (California): Apress, 2007, ISBN 978-1-59059-818-4.

[25] RIPKA, Pavel, *Senzory, jejich funkce, základní principy, motory* [online],
URL:<http://support.dce.felk.cvut.cz/roboti/index.php?id=organization>,
[cit. 2010-04-05].

[26] *Komunikace přes Bluetooth* [online],
URL:<http://mindstorms.lego.com/en-us/bluetooth/default.aspx>,
[cit. 2010-05-03].

[27] BISHOP, Owen, *Programming LEGO® MINDSTORMS® NXT*, Burlington (Massachusetts): Syngress Publishing, Inc., 2008, ISBN-13 781597492782.

[28] *Multimediální tutoriál NXT-G* [online],
URL:<http://www.ortop.org/NXT_Tutorial/index.html>, [cit. 2010-05-03].

[29] FERRARI Giulio, et al., *Programming LEGO Mindstorms in Java*, Rockland(Massachusetts): Syngress Publishing, Inc., 2002, ISBN 1-928994-55-5.

[30] STUBER, Jurgen, *LEGO NXT Programming* [online],
URL:<http://www.jstuber.net/lego/nxt-programming/index.html>,
[cit. 2009-10-26].

[31] *nxtasy.org – Server s novinkami ve světě LEGO® MINDSTORMS® NXT* [online],
URL:<http://nxtasy.org>, [cit. 2009-10-12].

[32] *nxtprograms.com – Server s návody na stavbu a programování robota* [online],
URL:<http://www.nxtprograms.com/index.html>, [cit. 2009-10-12].

[33] *Domovská stránka výrobce LEGO® MINDSTORMS® NXT* [online],
URL:<http://mindstorms.lego.com/en-us/community/NXTLog/default.aspx>,
[cit. 2010-05-03].

[34] PECINOVSKÝ, Rudolf, *Myslíme objektově v jazyku Java*, 2. vyd., Praha: Grada, 2008, ISBN 978-80-247-2653-3.

[35] *MSDN – Multitasking* [online],
URL:<http://msdn.microsoft.com/en-us/library/ms684259(VS.85).aspx>,
[cit. 2010-05-03].

A Obsah CD

Následuje obsah jednotlivých adresářů přiloženého CD.

CCR – zdrojové kódy příkladů CCR

DSS – zdrojové kódy služeb DSS

Fotky – fotografie stavebnice a robota

SimplySim – instalační soubor virtuálního prostředí SimplySim

Text – text této práce ve formátu pdf

VPL – zdrojové kódy příkladů VPL